Contract-based Specification of Embedded Control Systems

Oğuzcan Oğuz

Contract-Based Specification of Embedded Control Systems

O. Oğuz

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and
Computer Science, University of Twente

Robotics and Mechatronics group

CTIT Ph.D. Thesis Series No. 17-450
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands.

Title: Contract-Based Specification of Embedded Control Systems

Author: O. Oğuz

ISBN: 978-90-365-4497-9

ISSN: 1381-3617 (CTIT Ph.D. Thesis Series No. 17-450)

DOI: 10.3990/1.9789036544979

Copyright © 2018 by O. Oğuz, Enschede, The Netherlands.

All rights reserved. No part of this publication may be reproduced by print, photocopy or any other means without the prior written permission from the copyright owner.

CONTRACT-BASED SPECIFICATION OF EMBEDDED CONTROL SYSTEMS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. T.T.M. Palstra,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 16 februari 2018 om 12.45 uur

door

Oğuzcan Oğuz
geboren op 22 juni 1982
te Eskişehir, Turkey

Dit proefschrift is goedgekeurd door:

prof.dr. J.C. van de Pol, promotor

dr.ir. J.F. Broenink, co-promotor

dr. A.H. Mader, co-promotor

Promotiecommissie

Voorzitter

prof.dr. J.N. Kok	University of Twente
-------------------	----------------------

Promotor

prof.dr. J.C. van de Pol	University of Twente
--------------------------	----------------------

Co-promotoren

dr.ir. J.F. Broenink	University of Twente
----------------------	----------------------

dr. A.H. Mader	University of Twente
----------------	----------------------

Overige Leden

prof.dr.ir. B. Tekinerdogan	Wageningen University & Research
-----------------------------	----------------------------------

prof.dr.ir. H. Bruyninckx	KU Leuven
---------------------------	-----------

prof.dr. R.J. Wieringa	University of Twente
------------------------	----------------------

dr.ir. A.B.J. Kokkeler	University of Twente
------------------------	----------------------

Biricik Babama

Acknowledgements

The process of doing this thesis was a long one which eventually resulted in a manuscript I am proud to have produced. Here, I would like to express my gratitude to the people who supported me during this time.

First of all, I would like to thank all members of the thesis committee for their valuable assessment and comments. I would like to thank my assistant supervisor Angelika Mader for her trust in me and her creative input. I am grateful that you always encouraged me to find my own way. I would like to thank my assistant supervisor Jan Broenink for his valuable guidance and patience. You supported me greatly and were always willing to help me. I would like to thank my supervisor Jaco van de Pol for his inspiring attitude and constructive feedback. I feel very lucky that I had the chance to collaborate with you.

I am extremely grateful to all my family, friends and colleagues, from Philips and University of Twente, who, voluntarily or not, cared about this thesis as much as I did and supported me to the end. To name a few, I am grateful for the sparkling eyes, loud giggles and contagious energy of -in order of appearance- Tunç Perdahçioğlu, Sophie Dresscher, Asya Roza Karabudak, Aylin Yüce, Gökçe Sukaş, Ayça Yüce, Kemal Sukaş, Alisa İliş, Nora Perdahçioğlu and Deniz Kuban. I would not have known how to put those lego blocks together or how to sort shapes and colors without your guidance.

I would like to thank in particular to Kardelen Hatun as she was there for me all the way during this lengthy and, at times, rough period.

Most of all, I would like to thank my dear family, my father Ahmet Bircan Oğuz, my mother Gönül Oğuz and my lovely sister Nazlı Oğuz, for their practically proven near infinite support and trust in me. I love you; you mean the world to me.

Summary

In this thesis our overall aim is to provide a contract-based embedded system development process with an emphasis on specification and analysis. We specify components and layers via assume/guarantee contracts and use a contract algebra as the backbone to supply necessary operations to compose, refine and validate contracts. We employ various modelling formalisms, such as timed automata and hybrid automata to express contracts. The choice of the formalism is made according to the sort of behaviours of the component at hand. We employ appropriate model-based analysis methods, such as model checking and simulation, to support practical contract operations.

Our first contribution is a contract-based specification architecture and the associated specification process. The specification architecture is mode based where a system is specified with a number of operational modes and a mode-switching logic. The concrete formalisms used are based on timed automata for switching logic and hybrid automata for specification of individual modes of operation. Practical methods for composing and analysing hybrid automata-based contracts are defined over using hybrid observers which enable employing hybrid system simulation tools. To analyse timed automata-based contracts in isolation, model checking is employed. Specific validity criteria for mode and switching components are defined for each step of the specification process.

Our second contribution is a method to use timed automata-based specifications as observers in a hybrid system simulation tool (20-sim). We provide the associated implementation which employs Uppaal model-checking tool to execute such generic timed automata-based specifications alongside hybrid automata based-observer specifications. In the context of the described specification process, this enables validating contracts with assertions expressed in both timed and hybrid automata.

Our third contribution is a generic schedulability framework using model checking on timed-CSP models. The framework enables analysing multiprocessor schedulability of CSP models with non-preemptive fixed-priority tasks with variable execution times. We present a schedulability analysis workflow that describes how to utilize the proposed framework in a generic CSP-based design process. We also describe how to integrate the schedulability framework to a contract-based design process. Regarding the contract-based design of an embedded control software, we describe how a vertical timing contract can be formed by combining the control design contract and an execution platform contract. The resulting contract reflects the schedulability of the application expressed by the design contract on the execution platform described by the platform contract. Then, the proposed schedulability analysis framework can be used to check consistency of the timing contract.

Samenvatting

Het doel van dit proefschrift is een op contracten gebaseerd embedded systeem ontwikkelproces (met nadruk op specificatie en analyse) te introduceren. We specificeren componenten en lagen met behulp van assume/guarantee contracten en gebruiken een contract algebra als ruggengraat voor de benodigde operaties zoals compositie, verfijning en validatie van contracten. We gebruiken diverse modelleer formalismen zoals timed en hybride automata om contracten te specificeren. Formalismen worden gekozen aan de hand van het gedrag van de onderhanden component. We gebruiken geschikte model gebaseerde analyse methoden zoals model checking en simulatie om de eigenlijke contract operaties te ondersteunen.

Onze eerste bijdrage is een op contracten gebaseerde specificatie architectuur en het daarbij behorende specificatie proces. De specificatie architectuur is mode-based, terwijl een systeem gespecificeerd wordt met een aantal operationele modes en mode-switching logica. De specifieke gebruikte formalismen zijn gebaseerd op timed automata om logica te schakelen en hybride automata voor de specificatie van individuele operationele modes. Praktische methoden voor het samenvoegen en analyseren van hybride automata contracten worden gedefinieerd door hybride observers die het mogelijk maken om hybride systeem simulatie gereedschappen te gebruiken. Om timed automata contracten te analyseren wordt model checking gebruikt. Specifieke validatie criteria voor mode en mode-switching componenten zijn gedefinieerd voor iedere stap van het specificatie proces.

Onze tweede bijdrage is een methode om timed automata gebaseerde specificaties te gebruiken als waarnemers in hybride systeem simulatie tools (20-sim). We presenteren de bijbehorende implementatie die het Uppaal model-checking tool gebruikt om zulke generieke timed automata specificaties samen met hybride automata specificaties uit te voeren. In de context van het beschreven specificatie process maakt dit het mogelijk om contracten te valideren met asserties in zowel timed als hybride automaten.

Onze derde bijdrage is een generiek schedulability raamwerk dat model checking op timed-CSP modellen gebruikt. Het raamwerk maakt het mogelijk om de multiprocessor schedulability van CSP modellen met non-preemptive fixed-priority taken met niet constante executie tijden te controleren. We presenteren een schedulability analyse workflow die beschrijft hoe het voorgestelde raamwerk te gebruiken is in een generiek op CSP gebaseerd ontwerp proces. We beschrijven ook hoe het schedulability raamwerk geïntegreerd kan worden in een op contracten gebaseerd ontwerp proces. Aangaande het op contracten gebaseerde ontwerp van embedded regelsystemen, beschrijven we hoe een verticaal timing contract kan worden gemaakt door het control design en een executie platform contract te combineren. Het resulterende contract laat zien in hoeverre de applicatie, beschreven door het ontwerp contract, te scheduleren is op het executie platform, beschreven door het platform contract. Daarna kan

het voorgestelde schedulability analyse raamwerk gebruikt worden om de consistentie van het timing contract te controleren.

Contents

I	Prerequisites	xxvii
1	Introduction	1
1.1	Context	1
1.1.1	System Specification	1
1.1.2	System Design	3
1.2	Approach	4
1.2.1	System Specification	6
1.2.2	System Design	8
1.2.3	Main Contributions	10
1.3	Thesis Outline	10
2	Background	15
2.1	Model-Based Design	15
2.1.1	Modelling Hybrid Systems	16
2.1.2	Analysis Methods	17
2.2	Contract-Based Design	20
2.2.1	A Meta-Theory of Contracts	21
2.2.2	Assume/Guarantee Contracts	22
2.2.3	System Obligations	24
2.2.4	Practical Implementation of Assume/Guarantee Contracts with Tests	26
3	Case Study Description	29
3.1	The R2-G2Px Robot	29
3.2	Maze Navigation	31
II	Specification	33
4	System Specification with Modes	35
4.1	System Specification	35
4.1.1	A Model for System Specification	36

4.1.2	Quality Criteria for Requirements Specification	41
4.2	Modes	43
4.3	A Specification Architecture with Modes	45
4.3.1	Workflow for Specification	47
4.3.2	Validation of Contract-Based Specifications	50
4.4	Conclusions	51
5	A/G Contracts for System Specification	53
5.1	Assertions	54
5.2	Hybrid Automata	58
5.2.1	Open Hybrid Automata	61
5.3	Hybrid Observers	65
5.3.1	Testable Hybrid Observers	67
5.3.2	Assertions Expressed with Testable Hybrid Observers	73
5.4	Methods for Hybrid Observers	75
5.4.1	Implementing Observers in 20-sim	75
5.4.2	Simulating Observers	79
5.4.3	Syntax and Semantics of Hybrid System Simulation Tools	82
5.5	Validity of Assume/Guarantee Contracts	82
5.5.1	Correctness	83
5.5.2	Completeness	84
5.5.3	Consistency	85
5.6	Conclusions	85
6	Specification of Mode Behaviour	87
6.1	Specification of Mode Components	88
6.1.1	A Generic Mode Component	88
6.1.2	Case: Modes of the Robot	89
6.1.3	Case: Contract for the Line-following (m_2) Mode	92
6.1.4	Validity of Mode Contracts	96
6.2	Specification of Mode-Composition Component	100
6.2.1	Mode-Composition Component Architecture	101
6.2.2	Case: Contract for the Stopping (m_1) Mode	101
6.2.3	Validity of Mode-Composition Contract	102
6.3	Conclusions	104

7	Specification of Mode Switching	105
7.1	Generic Mode-Switching Contract	105
7.2	Assertions for Mode Switching	106
7.2.1	Timed Automata and Timed Observers	106
7.2.2	Using Uppaal on Simulation and Verification of Switching Contract	111
7.3	Case: Mode Switching of the Robot	115
7.4	Validity of Mode-Switching Contract	119
7.4.1	Correctness	119
7.4.2	Completeness	120
7.4.3	Consistency	121
7.5	Conclusions	122
8	Composition of System Specification Contract	123
8.1	Generic Overall System Contract	123
8.2	Using Uppaal-Tron for Timed Observers	124
8.2.1	Tron	124
8.2.2	Timed Observers in Uppaal	126
8.2.3	Connecting Tron and 20-sim	132
8.3	Case: Overall Robot Contract	135
8.4	Validity of System Contract	138
8.4.1	Correctness	138
8.4.2	Completeness	140
8.4.3	Consistency	141
8.4.4	Assessing Validity of Overall Robot Contract	144
8.5	Conclusions	153
III	Design	155
9	Contract-Based System Design	157
9.1	System design	157
9.1.1	Digital Control Design	159
9.1.2	Digital Control Model Semantics	160
9.1.3	Vertical Contracts	163
9.2	Conclusions	164

10 Schedulability Analysis for Vertical Contracts	167
10.1 Formulating Vertical Contracts for Schedulability	167
10.2 Schedulability Analysis with Timed-communicating sequential processes (CSP) Models	169
10.2.1 Timing Concerns of Systems Specified by CSP	169
10.2.2 Timed CSP and Process Analysis Toolkit (PAT)	172
10.2.3 Schedulability Analysis Workflow	175
10.2.4 Platform-Independent Process (PIP)	176
10.2.5 Instrumenting the PIP with Platform-Independent Timing	177
10.2.6 Specifying Hardware Mapping, Priorities and Execution Times	178
10.2.7 Adding Task and CPU Processes	179
10.2.8 Schedulability Analysis	183
10.2.9 Refinement Relation Between PIP and PSP	184
10.3 Case Study: Schedulability Analysis of R2-G2Px Robot Control Software	186
10.3.1 Robot Behaviour Specification	187
10.3.2 Platform-Independent Process (PIP) of Control Software	187
10.3.3 Construction of the Platform-Specific Process (PSP)	190
10.3.4 Analysis of the Platform-Specific Process (PSP)	191
10.4 Conclusions	193
 IV Reflections	 195
 11 Conclusions	 197
11.1 Results	198
11.2 Open Issues and Future Work	200
 A Frequently Used Symbols and Notations	 203
 Acronyms	 205
 Bibliography	 209

List of Figures

2.1	A hybrid bouncing ball model and it's movement.	17
2.2	Simulation of a hybrid bouncing ball model using Simulink	18
2.3	Venn diagram demonstration of assume and guarantee assertions of contracts C and C' , where $C' \leq C$	25
3.1	The R2-G2Px robot (Oguz et al., 2012)	29
3.2	The R2-G2Px robot on a referential frame.	30
3.3	The R2-G2Px robot on a sample maze	31
3.4	Transitions between the modes of operation of the R2-G2Px robot.	32
4.1	System and Environment	36
4.2	The four-variable model. Adapted from (Thompson et al., 2000).	37
4.3	The WRSPM reference model. Adapted from (Gunter et al., 2000)	39
4.4	System ports and variables	41
4.5	System ports and variables with controlled plant and the rest of the environment separated	41
4.6	Initial system decomposition.	47
4.7	Mode-composition component is decomposed into a number of modes.	47
4.8	System specification workflow.	48
5.1	An example for hybrid automata	59
5.2	Depiction of using a hybrid observer to test a system.	68
5.3	An example hybrid observer.	76
5.4	A source model and hybrid observer model connected in 20sim.	79
5.5	A simulation of the observer from Example 5.4.2.	79
5.6	An observer defined as a logical combination of three other observers.	80
6.1	Mode behaviour specification workflow.	88
6.2	Generic mode component with its ports.	89
6.3	mode-composition component is decomposed into five modes.	90
6.4	Ports between the m_2 component, and mode-switching component and environment.	91
6.5	Observer automaton for $A_1 \wedge A_2$	94

6.6	Observer automaton for A_3	94
6.7	Observer automaton for $A_4 \wedge A_5$	95
6.8	Observer automaton for G_1	95
6.9	Observer automata for G_2	96
6.10	20-sim block diagram for simulation of A_{m_2} and G_1	97
6.11	Simulation of A_{m_2} and G_1 with a behaviour including a bump.	98
6.12	Observer automaton for G_1 , modified for completeness.	99
6.13	Observer automaton for the assertion G_1	102
7.1	Ports of the generic mode-switching component.	106
7.2	A simple lamp modelled in Uppaal (Behrmann et al., 2006).	112
7.3	A timed automata template in Uppaal modelling accompanying automata for variable ν declared in Listing 7.2.	115
7.4	Ports of the mode-switching component.	116
7.5	The supplementary process for having urgent edges in Uppaal.	117
7.6	The Uppaal timed automaton for A_1 assertion of the switching contract.	118
7.7	The Uppaal timed automaton for A_2 assertion of the switching contract.	118
7.8	The Uppaal timed automaton for G_1 assertion of the switching contract.	118
8.1	Ports of the generic system component	124
8.2	Modeling of implementation under test (IUT) and environment in Uppaal. Bottom part of the figure depicts the real setup which includes the real environment (ENV) and the developed system (IUT). The top part depicts the environment and implementation models in Uppaal. Adapted from (Hessel et al., 2008)	125
8.3	Typical test setup with Uppaal and Tron. The Uppaal-Tron tester emulates the environment based on E and feeds the IUT with inputs. The outputs generated by implementation is monitored by the tester to check if it conforms to the implementation model S . Adapted from (Hessel et al., 2008)	126
8.4	Use of Uppaal-Tron only for implementation monitoring. IUT interacts with the real environment. The inputs and outputs of IUT are monitored by Uppaal-Tron to check if IUT conforms to the implementation model S . Adapted from (Hessel et al., 2008)	127
8.5	Uppaal monitor setup for open timed automata with input/output variables.	128
8.6	The passive process O	128
8.7	Example OTA_L	128

8.8 The supplementary process R defined for the variable declarations in Listing 7.3	131
8.9 A sample block diagram in 20-sim with a trace source and a timed observer, which is connected to Uppaal-Tron externally.	132
8.10 Overview of the connection between Uppaal-Tron and 20-sim.	134
8.11 An example simulation in 20-sim with an observer model that calls tron.	136
8.12 Output of 20-sim/Tron adapter for the simulation shown in Figure 8.11. The adapter output shows the delay and variable set calls made by 20-sim during the simulation.	136
8.13 Output of Tron engine for the simulation shown in Figure 8.11. Tron output shows Tron related settings and information about the given Uppaal specification.	137
8.14 A 20-sim simulation that is stopped at 30s due to call to Tron resulted in an error. The call resulting with error is a delay call made at 30s.	137
8.15 Output of 20-sim Tron adapter for the simulation shown in Figure 8.14. The adapter output shows the delay and variable set calls made by 20-sim during the simulation. Comparing with Figure 8.12, execution stops at the first delay call after the erroneous variable set call with values 3 and 0.	138
8.16 Output of the Tron engine for the simulation shown in Figure 8.14. Tron output indicates an error during the execution. Details of the state of the given Uppaal specification and input and output options at the point of error can be seen. For detailed explanation of the Tron output refer to (Larsen et al., 2009)	139
8.17 The observer model of overall system contract C_{sys} in 20-sim as the composition of observers for C_{sw} and C_{mc}	140
8.18 Conditions to disprove (a) or strengthen (b) consistency with respect to mode-set S	143
8.19 The Uppaal automaton to check if C_{sw} allows a switch from a set of modes $set1$ to another set $set2$	146
8.20 Conditions to disprove (a) or strengthen (b) consistency with respect to switching from mode-set S_1 to S_2	147
8.21 Conditions to check to show the consistency of switching between mode sets S_{m_2} and S_{m_1}	148
8.22 The complete 20-sim model used to assess consistency of C_{sys}	150
8.23 Simulation of implementation models to find maximum deviation of robot from the line in S_{m_2}	151
8.24 Simulation of implementation models to find the earliest time to get off-line in mode set S_{m_1} when started with the robot at maximum deviation from the line.	152

9.1	System decomposition for digital control design. System is decomposed into control software, and sensing and actuating components.	158
9.2	Digital control design models in 20-sim	161
9.3	A simple synchronous block diagram (synchronous block diagram (SBD))	162
9.4	Architectures of application contract C and platform contract P	165
10.1	A SBD model with two macro blocks and a switch block triggering the macro blocks	168
10.2	Architecture of a platform contract with two CPUs.	169
10.3	Schedulability analysis workflow.	173
10.4	External events of <code>ROBOT_CONTROL</code> process.	188
10.5	Process diagram showing the top level composition of the <code>ROBOT_CONTROL</code> process.	188
10.6	Process diagram showing the top level composition of <code>SEQUENCE_CONTROL</code> process.	189
10.7	Process diagram of <code>LOOP_CONTROL</code> process.	190

List of Tables

5.1	Mapping of hybrid observer parts to the sections and parts of 20sim observer implementation.	78
5.2	Four-valued truth table for $HO_C = (HO_1 \vee HO_2) \wedge \neg HO_3$	82
6.1	Textual requirements of the m_2 mode.	90
6.2	Classification of the requirements of the mode m_2 as assumptions and guarantees.	92
6.3	Textual requirements of the m_1 mode.	102
7.1	Textual requirements of mode-switching behaviour.	116

Listings

7.1	Declarations in Uppaal for the lamp model shown in Figure 7.2.	112
7.2	Declarations in Uppaal to be used by the accompanying automata in Figure 7.3.	115
7.3	Declarations in Uppaal to be used by the accompanying automata in Figure 8.8.	117
7.4	Initial declaration of <code>u_com</code> in Uppaal.	120
8.1	Declarations of variables for OTA_L	129
8.2	Global declarations for the supplementary process R in Figure 8.8 . . .	129
8.3	Local declarations in Uppaal for the supplementary process R in Figure 8.8	130
8.5	The declaration of <code>check_set</code> function used in Figure 8.19	146
10.1	A sample platform-independent process (platform-independent process (PIP)) called <code>SYSTEM</code> including 3 task kinds.	177
10.2	The process <code>SYSTEM</code> after it is converted into a timed process.	178
10.3	Sample Task and Mapped Process Attribute Arrays for <code>SYSTEM</code>	179
10.4	Description of <code>TASK</code> and <code>CPU</code> processes.	181
10.5	<code>SYSTEM</code> process after instrumented with <code>TASK</code> and <code>CPU</code> processes. . .	181
10.6	<code>PSP_SYSTEM</code> process after two time constrained processes in <code>CP1</code> and <code>CP1</code> are marked with <code>d_start.i</code> and <code>d_end.i</code> events.	183
10.7	Deadline count and values defined for <code>PSP_SYSTEM</code>	183
10.8	Descriptions of <code>DEADLINE</code> and <code>DEADLINES</code> processes.	184
10.9	Assertions for <code>PSP_SYSTEM</code>	184
10.10	Description of <code>ROBOT_CONTROL</code> process.	188
10.11	Description of <code>SEQUENCE_CONTROL</code> process.	189
10.12	Description of <code>LOOP_CONTROL</code> process.	190
10.13	Descriptions of <code>SEQUENCE_CONTROL</code> and <code>LOOP_CONTROL</code> processes after they are instrumented with periodic delays.	191
10.14	Task and mapped process attribute arrays specified for <code>ROBOT_CONTROL</code> . Schedulability analysis of <code>ROBOT_CONTROL</code> with this configuration fails to hold.	192

10.15 Description of PSP_ROBOT_CONTROL after it is instrumented to specify two time constrained processes in SEQUENCE_CONTROL and LOOP_CONTROL. PSP_ROBOT_CONTROL is combined with DEADLINES process. The schedulability assertion and the assertion for deadlock-freedom are formulated. 193

10.16 The modified mapped process attribute array which makes PSP_ROBOT_CONTROL schedulable. 193

Part I

Prerequisites

1

Introduction

1.1 Context

An embedded control system or an embedded system is a computer system with a dedicated objective in the physical world. An embedded system achieves its aims by monitoring and controlling the physical processes that lie in its environment.

With the advancements in size and capabilities of digital computers, embedded control systems have gained ubiquity in addressing the problems of industry and daily life. In industry, most of the processes are automated by embedded computers for the sake of safety and efficiency. In daily life, it is hard to find any task for people that has not been offered with an aid of a product embodying an embedded control system. An inevitable increase in complexity of embedded systems accompanies the increase in ubiquity as the systems become larger and connected to each other.

With the increased demand and complexity, embedded-system companies, in particular automotive, avionics and consumer electronics companies, are facing significant challenges regarding the functional correctness and reliability of their products and their time-to-market. Behaviour of an embedded system is defined by both computational and physical processes. Due to this hybrid nature of embedded systems, the concerns and the associated requirements of the embedded-system development process are different than those of non-embedded systems. In order to achieve efficient development processes that result in correct and reliable products, these concerns and requirements need to be identified and addressed.

The embedded-system development process is a mix of efforts of modelling, design, analysis, implementation and testing. Broadly speaking, at the very beginning of a development process, there are usually unorganized requirements and wishes. The development process starts with forming a system specification from the requirements. Then, model-based design and analysis is performed. Then, the design is implemented manually or automatically on an execution platform and tested.

1.1.1 System Specification

Any embedded-system development process starts with specifying the problem that the system to be developed is supposed to solve. For safety-critical embedded sys-

tems, completeness and consistency of system specification is vital to the safe and correct functioning of the end product. Accidents and major losses in safety-critical computer systems are usually due to the flaws in system specifications (Lutz, 1993; Leveson, 2000, 2004; Jaffe, 2011; Safeware Engineering Corporation, 2017).

Methods, formalisms and tools used in the system-specification process affect the quality of the resulting specifications. Although, the very starting point of all specification processes in general involves vague and informal statements, the end product of a specification process should be expressed in an exact manner without leaving room for different interpretations. Vagueness in system specification prevents the validity of the specification from being reliably assessed. Also, correctness of a design cannot be assessed with respect to a vague specification. However, exactness should not be confused with over-specification. Any system specification should be abstract in the sense that it allows for different designs steered by the decisions related to the constraints that do not affect the system's functionality such as development cost, third party suppliers, implementation and platform constraints, etc. In this regard, the formalisms used to express system specifications should be exact and should allow for abstract specifications (non-determinism).

Formal specification languages with exact syntax and semantics are suitable for this purpose. However, the specification process is not merely deriving a mathematical description of the desired system behaviour. The specification has to be investigated and analysed for validity and correctness of designs has to be assessed against the specification. Therefore, the specifications have to be executable. Furthermore, in order to handle complexity, the selected formalisms should enable component-based design. Also, they should be able to describe both the digital and the physical nature of the embedded systems. In this direction, hybrid system modelling formalisms, such as hybrid automata, are suitable.

Methods and steps to capture and structure the system requirements, and to construct the system specification using the selected formalisms are also essential to achieve good quality specifications. Appropriate specification architectures, much like the design patterns in software design, can act as templates for solutions to commonly occurring specification problems. Like any other system, an embedded system is developed to perform within specific environments. For this reason, the system specification should be structured such that the assumptions on the environment are explicitly stated.

Last but not least, the criteria that define the properties of specifications with good quality should be defined. The quality criteria and the methods to assess them should be tailored to the employed formalisms and architectures to enable effective and automatic/semi-automatic assessment of specification quality.

In summary, the list of properties of specification formalisms and methods to get specifications with good quality is as follows:

- Exact but abstract specifications
- Executable/analysable specifications, not just mathematical descriptions
- Support for hybrid behaviour

- Explicit assumptions on environment
- Explicit quality/validity criteria, such as completeness and consistency, and methods to check them
- Support for component-based descriptions
- Use of appropriate architectures

1.1.2 System Design

Once the system specification is constructed and validated, the development process proceeds with system design. The aim of system design is to realize the specification within the enforced design constraints, such as time, cost, (reuse of) existing/off-the-shelf components, etc., that are separate from the system's desired functionality. Seamless integration of the specification and design processes is vital to support the iterative nature of the design process. Through the design process, the design iterations should be able to be compared to the whole or the parts of the system specification. For this, given a specification, the correctness criteria for the design and the methods for assessing them should be well defined.

The two methods that make a complex design task manageable are decomposition and abstraction. Component-based design enables the decomposition of a whole system into manageable pieces. Layered design enables performing design activities at a layer of abstraction where the building blocks are provided by the lower layer and details of the implementation of the services of the lower layer are abstracted away at the higher layers. In a design process, development of components by different parties and reuse of existing or off-the-shelf components require adequate specification of individual components. All in all this calls for a specification theory that defines how the components are specified and how they are composed.

During the design, ability to support multiple viewpoints is also important. Representing the desired behaviour of a system or a component pertaining to all viewpoints in a single encompassing formalism is not practical or possible. Behaviours of a component or a system are represented with respect to different viewpoints, such as functionality, timing, resource utilization, safety, etc., using the appropriate formalisms. Design and analysis of different viewpoints might be done by different teams with different skill sets and tools. A design process should be able to handle different viewpoints.

Here we list the mentioned properties of a design process:

- Seamless integration of specification and design through refinement relations
- Support for an integrated component-based and layered design/development: decomposition and abstraction, and development by multiple parties
- Support for expressing and analysing behaviours of various nature and multiple viewpoints

In this thesis our main goal is to describe an embedded-system development process with the stated properties regarding system specification and design. In the next section, we briefly describe the existing methods addressing the concerns stated above.

Then, we describe our approach to embedded system specification and design which integrates and extends the current methods.

1.2 Approach

The two key methods that are in use and have the potential to address the concerns we raised in Section 1.1 are model-based design (MBD) and contract-based design (CBD). The overall approach in this thesis and individual contributions are based on these two methods. Here, we summarize how MBD and CBD propose solutions to our concerns. Then, we introduce our approach based on MBD and CBD in Subsection 1.2.1 and Subsection 1.2.2, focusing on system specification and system design, respectively. More details on MBD and CBD that are essential to our discussion are presented in sections 2.1 and Section 2.2 of Chapter 2.

Model-based design is a methodology for developing embedded systems that is based on constructing and analysing models for the system and its environment. In a model-based design process, starting with the system requirements, the system and its environment are modelled. The core idea is, through modelling and analysis, to gain insight into the behaviour of the system in order to validate the requirements and the proposed designs. Modelling activities can be focused on different viewpoints, components or abstraction levels of the design. Models can also be used to generate implementations and support testing.

Modelling formalisms and tools to construct and analyse models are at the heart of model-based design. Formalisms with precise semantics prevent surprises in specification and design processes. The employed formalisms are chosen according to the modelling task at hand. Different viewpoints are supported by specialized formalisms and tools. Modelling of system-level functions, the physical system and control laws, software design, execution platform require different formalisms and tools. Especially, modelling of hybrid behaviour that is essential in the context of embedded systems is supported by hybrid-system formalisms and tools.

A model is almost useless if it can not be executed to analyse its behaviour. Analysis of models enables validation¹ of system specification and design. The methods for analysis can be categorized into simulation-based methods or formal verification based methods. Simulation is the method of mimicking a model's behaviour over a period of time. Formal verification is assessing correctness of an abstract mathematical model with respect to a property by exploring all the states of the model. Methods of analysis depend on the capabilities of the tools at hand and method's practicality for a given problem. While simulation cannot provide exhaustive analysis as formal verification does, formal verification is not always possible or practical due to decidability and scalability issues.

¹ Safety-critical software development guidelines and standards define *verification* as the process of ensuring that software being developed satisfies functional and other requirements (Jacklin et al., 2004). On the other hand the process of ensuring that the requirements are the correct requirements and that they are complete is called *validation*. In our work, we use the term validation for both purposes and reserve the term verification for formal verification activities supporting validation.

Regarding model-based design, individual modelling formalisms and tools used in the industry today are strong in their specialized domains, and they collectively cover almost all aspects of system design: SysML (Friedenthal et al., 2009) and AADL (Feiler et al., 2006) for system level modelling, Modelica (Fritzson, 2003) and MATLAB Simulink (The Mathworks Inc., 2016a) for physical system modelling and control law design, and Scade (Esterel Technologies, 2016) for detailed software design. Current state of art of applying MBD for embedded systems is that individual sectors or companies define their own development workflows that are tailored for their particular interests integrating a number of modelling tools (Broy et al., 2010). There is a need for a meta-level, encompassing methodology that is able to integrate individual model-based methods and supporting overall specification and design concerns we pointed out in Section 1.1 such as component-based and layered designs, handling of different viewpoints and separation of system's assumptions from its guarantees.

When such a methodology is first defined and understood independent of formalisms and tools, it can be tailored to integrate different formalisms and tools and different development processes. Contract-based design (Benveniste et al., 2012) is a step towards this aim. Key elements of CBD are contracts and an algebra for manipulating contracts. A contract is a specification of a component's behaviour with two key properties:

1. A contract is abstract, it specifies only the necessary amount of information about a component's behaviour.
2. A contract separates the responsibilities of a component from that of its environment. A contract not only specifies the guarantees of the component but also its assumptions about the environment.

The contract algebra presented in (Benveniste et al., 2012), defines contracts and contract operations independent from a concrete formalism. The meta contract algebra is made concrete by fixing the formalism of the contracts and the definition of the associated operations. In a contract-based design process, all of the viewpoints and components, from specification and design are represented by contracts. During the process, contracts are combined and/or related using the operations and relations defined in the contract algebra.

CBD naturally supports component-based and layered designs by the operations for combining contracts. Separating responsibilities of component and of its environment allows for development by different parties. It supports a development process where different formalisms and viewpoints are supported. Through refinement relations, specification and design processes are seamlessly integrated.

Our technical discussion in this thesis is presented in two parts.

In Part II of this thesis we focus on system specification. Using an integration of CBD and MBD methods, we aim to provide a system-specification approach with the properties stated in Section 1.1. In this direction, we provide a CBD-oriented system-specification process for embedded systems based on a mode-based architecture. For the contracts used in the specification process we provide concrete formalisms, validity criteria, practical methods to perform contract operations and assess validity of

contracts. In Subsection 1.2.1 further details of our specification approach are provided.

In Part III of this thesis we focus on system design and analysis. Using an integration of CBD and MBD methods, we aim to provide parts of a system design approach with the properties stated in Section 1.1. We provide the description of how a specification contract can be refined by a design contract via conventional controller design. We describe the use of vertical contracts to obtain a contract from the design contract from the schedulability point of view. To analyse the schedulability, we provide a generic schedulability analysis framework based on model checking using timed CSP. In Subsection 1.2.2, further details of our design approach are provided.

1.2.1 System Specification

In Part II of this thesis we describe a mode-based system-specification architecture and a specification workflow that employs assume/guarantee (A/G) contracts. We start with defining a generic system specification model similar to the four-variable model (Parnas and Madey, 1995) and the WRSPM model (Gunter et al., 2000). Different than those models, our specification model is aimed at specifying systems as a complete black-box without stating controllability and visibility of environment variables, as determining the variables to observe and influence is part of the design. Also our specification model is contract-based, clearly distinguishing responsibilities of environment and the system to be developed.

A mode-based specification architecture is composed of system modes and a mode-switching logic. Such an architecture enables partitioning of system specification into main system functions that can be specified independently which results in higher quality specifications (Maraninchi and Remond, 1998; Miller, 1998; Parnas, 2010).

Our specification is contract-based where all the modes of the system and the mode switching logic are specified with contracts using an A/G formalism. In this direction, we adopt the meta-theory of contracts presented in (Benveniste et al., 2012) which defines the contracts and contract algebra independent of any concrete formalism to express behaviour. To make the meta-theory effective, we define a concrete formalism to express behaviour in contracts. The concrete formalism is capable of expressing hybrid behaviour which is essential in system specification. Using the defined concrete formalism we are able to construct hybrid contracts. In order to enable use of hybrid contracts, i.e. expressing contracts and enable practical contract operations, in hybrid system simulation tools we propose formulating contracts using observers. An observer is an assertion that a simulated trace can be tested against in a simulation tool. We state the conditions for hybrid contracts to be online-testable and we demonstrate use of hybrid contracts in a physical systems modelling and simulation tool, 20-sim (Broenink, 1999; Controllab Products, 2017).

For specification of system modes, we use the described observer-based hybrid contracts. We state the specific validity criteria for mode contracts and employ simulation methods to assess the validity. For specification of mode switching logic, we construct contracts using timed-automata models in Uppaal (Behrmann et al., 2006). We state

the specific validity conditions for mode-switching contracts and perform the validity assessment in Uppaal using simulation and formal verification.

The overall system contract is defined as the composition of mode and switching contracts. In order to analyse the overall system contract, which is composed of observer-based hybrid contracts expressed in 20-sim tool and non-observer contracts expressed in Uppaal, we present a method of simulation of observer-based contracts and non-observer contracts together. This extension is implemented using the model-testing extension of Uppaal called Tron (Hessel et al., 2008; Larsen et al., 2009). For the validity of the overall system, we state the specific criteria and assess the validity in 20-sim. Throughout the whole discussion we demonstrate the specification steps on a case study of a two-wheeled robot.

In literature, to express mode-switching logic, formal specification languages are used, whether they are state-machine based formalisms (Leveson et al., 1994; Harel et al., 1990; Jahanian and Mok, 1994; Paynter, 1996; Maraninchi and Remond, 1998; The Mathworks Inc., 2016b) or tabular representations (Parnas, 2010; Faulk et al., 1992). In order to express system's behaviour when it is at a particular mode, the tools Statecharts (Harel et al., 1990), Mode-automata (Maraninchi and Remond, 1998), Stateflow (The Mathworks Inc., 2016b) use dataflow languages. In these tools modes of the system are associated with equations or tasks defined on the inputs and outputs, which define the mode behaviour. These tools are primarily design oriented languages so they are not really aimed towards specification of a system. They lack the ways to specify behaviours in an abstract way.

The formalisms specifically aiming system specification (Leveson et al., 1994; Parnas, 2010), on the other hand, do not prescribe explicitly any formalism or method to specify the mode behaviours of the system. They focus on specifying just the mode-switching logic. They do not explicitly specify behaviours of individual modes, nor they look at the combination of modes with each other and with the switching logic.

Above all, different than mentioned mode-based methods, our specification approach is contract based, treating individual modes and the mode-switching logic as specification components and specifying them using A/G contracts. Coupled with various formalisms and analysis methods, CBD have been extensively used for system design (Boström et al., 2007; Boström, 2011; Benveniste et al., 2008, 2009, 2012, 2015; Benvenuti et al., 2008a,b, 2014; Nuzzo et al., 2014, 2015; Bauer et al., 2012). However, a CBD process aimed at all practical and methodological aspects of hybrid system specification is new.

In (Benveniste et al., 2012, 2015), a detailed case of requirement specification for a car parking system is presented. However, it does not address hybrid behaviour which is essential in specifying embedded systems.

In (Bauer et al., 2012), similar and complementary to (Benveniste et al., 2012), a meta-theory of contracts is presented. Authors start with an axiomatization of generic interface theories and derive an A/G contract theory. Instantiation of the meta theory is based on a modal interface theory (Raclet et al., 2011), where a behaviour is speci-

fied using modal transition systems (Larsen and Thomsen, 1988) with *may* and *must* transitions. However, modal interfaces do not support hybrid system specification.

In (Benvenuti et al., 2008a,b, 2014), use of hybrid automata to formulate A/G contracts are demonstrated. In (Benveniste et al., 2008, 2009) use of hybrid automata and hybrid observers to formulate A/G are demonstrated. Neither of those methods focus on specification and specification validation and they only describe use of formal verification for analysis, leaving out the testability of observers and widely adopted simulation methods.

In (Nuzzo et al., 2014, 2015), requirement formalization and validation using contracts are described. Temporal logic and hybrid automata are proposed as formalisms to formulate contracts. However, requirement validation has not been elaborated.

In (Boström et al., 2007; Boström, 2011), A/G contracts are used to specify behaviour of Simulink components and compositionally verify control designs with many components. Conditions of correctness of individual components and composition of them with respect to their contracts are defined and formal verification and test based validation methods are described. However, these works do not feature a full contract algebra where viewpoints and conjunction are also addressed. Furthermore, these works aim only at Simulink whereas our work defines and employs hybrid contracts independent of any tool.

We think that having a proper template for specification architecture, a mode-base architecture in our case, and an associated workflow is vital in structuring system specification. This has not been addressed within the context of CBD. In system specification, validation criteria for the specification is crucial. In (Benveniste et al., 2012), generic validation criteria for CBD requirements specification, inspired by adopted requirement specification criteria in literature, are provided. However, these generic validity criteria are too weak since the criteria are defined on checking only for non-empty sets of behaviour. For a particular specification architecture, it is possible to define stronger and eventually more useful validity criteria, i.e. the criteria that check more meaningful, architecture specific behaviour. We provide tailored validity criteria for the proposed mode-based specification architecture.

Also, our presentation of using observer-based and non-observer assertions within a contract and simulation-based analysis of both sorts of assertions for the validation of the contract is new.

1.2.2 System Design

Given a validated system specification, expressed as an A/G contract, the development process proceeds with system design and implementation. Since an embedded system's objectives are defined in the environment, in the design process, the next step is control design. In Part III of this thesis we describe, given a system specification contract, how one can proceed with control design and timing analysis of the control implementation. Control design using contracts has been addressed in (Boström et al., 2007; Boström, 2011; Benvenuti et al., 2008b; Nuzzo et al., 2014, 2015; Sangiovanni-Vincentelli et al., 2012). In contrast to these works, we do not adopt a contract-based

control design. Component-based synthesis of controllers from specification is also an option for system specifications with restricted dynamic behaviour, such as discrete or timed systems (Kuijper and van de Pol, 2009). Instead, we presume, as it is mostly the case in the industry, that the control design is performed in a model-based manner using a hybrid system simulation tool such as Simulink or 20-sim. We describe how a model resulting from such a control design process can be assessed against an observer-based specification contract to see if it implements the system specification. Our approach is thus compatible with traditional model-based control design processes.

Correctness of a digital control design depends on the timely execution of the control laws it consists of. For this reason, in a CBD process aiming embedded systems development, timing concerns of control design should be addressed. For a particular design contract, we propose obtaining a contract for the timing viewpoint, on which the timing of the control law execution can be analysed. Consistency of a timing contract is defined as the schedulability of the execution tasks within the contract.

To analyse the schedulability on a timing contract, i.e. to check if the timing contract is consistent, we provide a schedulability analysis framework based on model checking on timed-CSP models. The schedulability framework is composed of task and processor models defined as timed-CSP processes in the PAT tool (Sun et al., 2009a). These framework structures are high-level abstractions that define the scheduling behaviour in the system. The supported scheduling scheme is fixed-priority non-preemptive scheduling. To analyse schedulability of a real-time application, the CSP model of the application is constructed. Then the combination of the application and the platform is constructed by specifying the mapping of the application onto the execution platform. In the constructed model, the tasks are assigned with fixed absolute priorities and mapped to processors. Best and worst case execution times for the tasks are also incorporated to account for variations in the execution times of the tasks. Then, on the composed model, the deadlines required to be met by the system are checked using model checking on timed CSP according to the CSP model of the application, priorities, execution times and mapping of the tasks, and the fixed-priority non-preemptive scheduling scheme.

Our schedulability analysis framework is generic so it can be employed as well in a development processes that do not employ contracts and rely on CSP-based tools and programming languages for design, analysis and code generation. In classical real-time scheduling theory, there are well established traditional schedulability analysis methods to check schedulability guarantee for various preemptive or non-preemptive scheduling schemes with periodic and/or sporadic tasks with fixed or dynamic priorities (Buttazzo, 2004; Burns, 1994). However, traditional schedulability analysis methods are seriously challenged in case of multiprocessor systems and the allowed task structures are restricted to periodic or sporadic tasks with simple precedence relationships.

Real-time model checking, which our approach is based on, is an alternative way of performing schedulability analysis. With these methods multiprocessor schedulabil-

ity analysis is inherently possible. Task arrival patterns and task synchronizations are less restricted which promotes the applicability of the analysis. Also, the analysis results are less pessimistic and provide higher processor utilization since systems and task arrivals can be modelled in more detail. The prominent formalism in model-based schedulability analysis is timed automata (Alur and Dill, 1994) or the task automata (Fersman et al., 2007), an extension of timed automata with tasks. Uppaal real-time model checker (Behrmann et al., 2004) is mostly employed in the implementations.

Uppaal-based schedulability analysis frameworks (David et al., 2010; Mikučionis et al., 2010) and the TIMES tool (Amnell et al., 2004) model the scheduling problem with task and resource models that synchronize with each other, and the related additional data structures that store task and resource data. Regarding our problem, schedulability analysis of systems described in CSP can be done using one of the Uppaal-based solutions mentioned above. This requires transforming the CSP models into Uppaal systems in accordance with the adopted Uppaal-based schedulability framework. Then the schedulability of the system can be verified by reachability checking in Uppaal. In case of an unschedulability result, a reverse mapping from the witness trace generated by Uppaal to the CSP model should be provided to locate the cause of unschedulability in the CSP model. However, defining such a transformation and a reverse mapping would be a tedious approach.

Instead, our approach is based on dense-time model checking directly on timed-CSP models using the PAT tool. We also present an associated design workflow in which the schedulability analysis is embedded in the process-oriented design of the system where a scheduled and timed-CSP model is constructed. On this model, besides the schedulability, it is also possible to formally verify other properties of the system, such as liveness properties and deadlock freedom, using the PAT tool.

1.2.3 Main Contributions

The main contributions of this thesis are as follows:

- A mode and contract-based specification architecture and the associated specification process. The concrete formalism used is based on hybrid automata and practical methods working with it are supplied. Specific validity criteria are defined for each step of the specification.
- Use of specifications formulated in timed-automata formalism in Uppaal as observers in 20-sim simulations, which enables executing generic timed automata-based specifications alongside hybrid automata-based observer specifications.
- A generic schedulability framework using model checking on timed-CSP models. We describe the use of the schedulability framework in the generic context and as integrated in a contract-based design process.

1.3 Thesis Outline

The outline and the contributions of individual chapters are as follows:

Part I: Prerequisites

In *Chapter 2*, we provide the background on contract-based design. Our main reference is (Benveniste et al., 2012), in which a theory of a contract algebra is presented. The contract algebra defines the form of contracts and various operations defined on contracts. It is generic enough to be employed in every step of a development process, enabling a full contract-based process.

In *Chapter 3*, we describe our case study on which we will demonstrate our specification and design approaches. The case is defined on a two-wheeled robot designed to traverse a labyrinth.

Part II: System Specification

In *Chapter 4*, we describe a mode-based system specification architecture and a specification workflow that employs assume/guarantee contracts. In particular, we show that use of system modes and mode-switching logic in system specification is suitable for partitioning main system functions. We show that a mode-based specification architecture and the associated specification process can be realized by using assume/guarantee contracts. Use of a mode-based partitioning and assume/guarantee contracts in system specification results in higher quality specifications which can be fed into a contract-based design process.

In *Chapter 5*, we describe a concrete formalism of assume/guarantee contracts that is suitable for system specification and the methods for using the defined contracts in practice. We define a concrete assume/guarantee contract formalism that supports hybrid behaviour, which is required in system specification phase due to the hybrid nature of the domain. Then, we show that one can use observers in order to express contracts in a way that is suitable to employ simulations for analysis. Having a formal definition for contracts with hybrid behaviour provides tool-independent and precise semantics. On the other hand, use of observers enables hybrid system simulation tools to be used effectively in practice on analysing validity of contracts. In combination, having precise semantics that is suitable for specification and the accompanying practical methods enables better quality specifications.

In *Chapter 6*, we describe how the modes of a system can be specified using the presented assume/guarantee contract formalism. The assume/guarantee notion of the employed contract formalism allows us to specify and validate the system modes independently, which is crucial for the proposed mode-based specification architecture. We show that a mode can be specified with an assume/guarantee contract that is composed of a collection of observers. Our approach allows formulation of individual observers directly from the requirements, which reduces the complexity of obtaining a formal specification. Combining individual observers into contracts which can be readily analysed with simulation comes cost-free. We show that the ability to employ simulations on the contract-based mode specifications enables to discover and fix validity issues that might be inherent to the initial requirements or introduced during the contract formulation.

In *Chapter 7*, we show how to specify mode switching behaviour of a system with an A/G contract. In order to express switching behaviour, using timed behaviour, a restricted form of hybrid behaviour, is sufficient. In this direction, we present definitions for timed automata and timed observers to express contracts with timed behaviour. Due to the decidability of properties of timed automata and the availability of an efficient model checking tool, Uppaal, we use formal verification to assess validity of a switching contract. A switching contract is more complex than a mode contract. Therefore, formulating the switching contract with many observers, as in one per requirement item, or with a single large observer is error prone. Also, it is harder to ensure completeness of a such a switching contract. We propose using non-observer assertions which can be composed in Uppaal with certain restrictions on the ports restricted by individual assertions. The composition is a network of timed automata that can be directly simulated and formally checked for properties in Uppaal. We define validity criteria and stated the points to consider while assessing these criteria. We demonstrate on the case that simulation and formal verification in Uppaal can be employed to check validity of a switching contract.

In *Chapter 8*, we present the methods to construct and validate the overall system contract. The overall system contract is the composition of mode contracts and the switching contract. The assertions in the constituent contracts of the overall system contract can be expressed in 20-sim as hybrid observers or in Uppaal as non-observer specifications. In order to be able to execute both sorts of assertions together, i.e. the whole system contract, we present a method to execute Uppaal specifications in 20-sim. The method employs the model-based testing extension of Uppaal called Tron. We describe how an assertion expressed in Uppaal can be supplemented with helper processes that enable using the assertion as an observer. Similarly, at the 20-sim side, we describe how one can model an observer that makes external DLL calls to send variable values and delays to the Tron tool to check if the sent values conform to the associated Uppaal model. We describe how we implemented the connection between Tron and 20-sim.

Regarding contract validation, we present the correctness, completeness and consistency criteria adapted for the overall system contract. In particular, we extend the consistency criterion to cover for the consistency of individual mode sets and the switches between the mode sets. We provide the intuition and formal definitions of these extensions. On the robot case, we demonstrate how the extended consistency criterion can be checked on a system contract. We show that the assume-guarantee nature and the executability of overall system contract enable reliably validating the consistency using a hybrid system simulator, 20-sim in our case.

Part III: System Design

In *Chapter 9*, we describe, for a given system specification contract, how to proceed with the system design to realize the system specification. The first step in the design process is performing digital control design using a control design tool. The result is a control design model that implements the specification. While performing the control

design in a simulation tool, we show that an observer-based specification is useful to assess if the control model being developed implements the system specification contract. We show how the control model resulting from the control design process can be used to construct an A/G design contract which refines the specification contract and the contract-based design process can proceed on this design contract.

We also explain the notion of vertical contracts which are used to relate two contracts at successive abstraction levels, e.g. mapping an application contract to the contract for the execution platform. Vertical contracts are used in Chapter 10 to analyse schedulability of a digital control implementation of design contract on an execution platform.

In *Chapter 10*, we describe how a vertical timing contract can be formed by combining a control design contract and a platform contract. The resulting contract reflects the schedulability of the application expressed by the design contract on the execution platform described by the platform contract. Then, we present a schedulability analysis framework based on timed CSP which can be used to check consistency of the timing contract. However, the schedulability analysis framework is generic so it can be employed as well in development processes that do not employ contracts and rely on CSP-based tools and programming languages for design, analysis and code generation.

The schedulability analysis framework is based on dense-time model checking in the PAT tool to check the schedulability of control systems modelled in CSP. Analysis is based on dense-time model checking on timed-CSP models. The supported scheduling scheme is non-preemptive fixed-priority multi-processor scheduling. Our framework enables analysing multiprocessor schedulability of systems modelled in CSP involving complex task synchronizations and variable task execution times that traditional schedulability analysis methods fall short on solving. Since our framework is based on model checking and the CSP model of the control system is incorporated in the analysis, the results are non-pessimistic. We also present a schedulability analysis workflow associated with the proposed schedulability analysis framework. The workflow defines the successive steps to construct and analyse a timed platform-dependent CSP model scheduled according to fixed-priority non-preemptive scheduling scheme from an untimed and unscheduled, platform-independent model of the control system.

2

Background

In this chapter, we provide an overview of the two methods, namely model-based design (MBD) and contract-based design (CBD), upon which we build our system specification and design efforts (Section 1.2). In Section 2.1, we briefly introduce model-based design. We mention the main idea behind MBD, and in general, the sort of models and the analysis methods employed in embedded system development. The actual model formalisms and analysis methods we use are introduced in Parts II and III.

In Section 2.2, we introduce the idea of contract-based design. First, we introduce a meta-theory of contracts describing what a contract as a component specification entity should capture, and what operations and relations on contracts are essential to perform a contract-based design process. Then, we present assume/guarantee (A/G) contracts as an instance of the meta-theory where a component is specified as a couple of assertions describing its legal environments and valid implementations. We define the operations and relations on A/G contracts. Lastly, we mention how tests can be used to express A/G contracts to make it effective for infinite domains.

2.1 Model-Based Design

Model-based design is a methodology for developing embedded systems based on constructing and analysing models for systems and environments that host systems. A *model* of a system is a description of certain aspects of the system. The aim of constructing a model is to gain insight into the aimed properties of the system. Models are always abstract in the sense that they omit most of the details of a system focusing only the aimed properties. How accurately a model represents the aimed properties of a system defines its fidelity. Throughout this thesis, we focus on mathematical models that enable systematic analysis, which is the way to expose what is captured in a model regarding the modelled system.

One of the main uses of model-based design in embedded system development is the ability to support layered design and component-based design, the two methods stated in Section 1.1 to be the enablers in coping with complexity of a design. In this respect, modelling languages allow for modelling at different levels of abstraction, such as system level languages SysML (Feiler et al., 2006) and AADL (Frieden-

thal et al., 2009), supporting layered-design. Modelling languages, such as MATLAB Simulink (The Mathworks Inc., 2016a), Scade (Esterel Technologies, 2016) and UML (Object Management Group (OMG), 2017), support component-based modelling by defining component and interface structures.

2.1.1 Modelling Hybrid Systems

Embedded systems are composed of physical parts such as a physical plant to be controlled, and logical parts such as a digital system running an algorithm to control the physical plant. One of the strengths of model-based design is the support of early virtual system integration, i.e. the integration of physical and logical parts, through modelling and analysis. Modelling physical and logical parts of an embedded system requires different modelling semantics in the manner that modelling physical phenomena requires expressing continuous dynamics whereas modelling logical or computational phenomena requires discrete semantics. From this perspective, an embedded system is a hybrid system, and efforts to model the hybrid nature of an embedded system is called hybrid system modelling. In the scope of this thesis, our aim is to employ MBD in system specification and early design following the specification. Specifying overall desired behaviour of an embedded system asks for a hybrid view of the system. For this reason, we focus on model-based design efforts focusing on hybrid system modelling and analysis.

Each hybrid system modelling language has an underlying semantics to represent hybrid behaviour of a system. Even though the semantics vary from one language to another, the general concepts are shared (Carloni et al., 2006). Here, we briefly and informally describe the generic idea behind hybrid system semantics. In Chapter 5 and Chapter 7 we provide full formal definition for the hybrid system semantics that we employ.

A hybrid system behaviour is a composition of discrete and continuous dynamics. Given an hybrid system with a set of variables, a behaviour of such a system is a mix of continuous flow and occasional discrete jumps of variable values. Continuous dynamics or flow in a hybrid system behaviour are governed by ordinary differential equations defined on the system variables. Discrete dynamics or jumps in a hybrid system behaviour are defined by the discrete states of the system and the transitions between them. Typically, a discrete state defines a set of differential equations that define the flow on the system variables when the system is at that state. Discrete jumps are triggered by external input or by the continuous evolution itself. As well as changing the continuous dynamics of the system, a discrete jump can also assign new values to the variables with which the system will resume its continuous flow at the new state.

In Figure 2.1a, a hybrid model of a the classical hybrid system example bouncing ball is shown. The model represents a bouncing ball being dropped from a predefined height. The ball hits the ground after a certain time, loses energy and bounces back into the air and then starts to fall again. Figure 2.1b depicts the balls height change with time. In the single discrete state of the hybrid model, represented with the circle, motion of the ball is governed by the differential equations $\dot{h} = v$ and $\dot{v} = g$, where h

Figure 2.1: A hybrid bouncing ball model and its movement.

is ball's height, v is its velocity, and g is the gravity constant (the mass of the ball is assumed to be 1kg).

The arrow left to the state represents the discrete transition taken when the ball reaches the ground. The invariant $h \geq 0$ enforces that the ball always takes the transition when it reaches the ground. The guard $h == 0$ of the transition, which models the bouncing, ensures that bouncing happens after reaching the ground and not before. The reset condition $v = -cv$ accounts for the loss of energy, where $0 \leq c \leq 1$ is a constant.

With such a hybrid system model, the aim would be to gain insight into the ball's bouncing motion and potentially prove or disprove a hypothesis through analysis of the model. In fact, without to the point analysis or execution, having a mathematical model is not useful. Regarding the hybrid system models, there are mainly two kinds of analysis methods. These are simulation and formal verification. Each method has its capabilities and limitations.

2.1.2 Analysis Methods

Simulation is the method of mimicking the behaviour of a system with the provided initial conditions and external inputs over a period of time. The data obtained during a number of simulations is used to gain insight to the model's behaviour under certain conditions, i.e. initial conditions and external inputs provided during the simulation. Prominent hybrid system simulation tools include MATLAB Simulink (The Mathworks Inc., 2016a) and Dymola (Dassault Systemes, 2017). In Figure 2.2, a simulation run of the bouncing ball model in Simulink is shown. In the figure, values obtained for

velocity v and the vertical position h of the ball in a simulation run for 25 time units are depicted. It can be observed from this simulation run that for the given initial conditions and the parameters, the velocity of the ball becomes zero in about 20 time units.

Hybrid system simulation tools employ advanced numerical algorithms to simulate a system. Usually, continuous behaviour is approximated using integration algorithms by stepping through a time interval, called an integration step, and calculating the integral of the derivatives in the differential equations of the model. Discrete behaviour is handled by monitoring and executing discrete events such as state transitions during the continuous simulation. For an overview of the numerical simulation techniques, please refer to (Frehse, 2015). For a comparative analysis of prominent hybrid system simulation languages and tools, please refer to (Carloni et al., 2006).

Figure 2.2: Simulation of a hybrid bouncing ball model using Simulink

In the hybrid systems context, formal verification is the method of proving or disproving a property for all possible modes of a formal hybrid system model. The idea is very attractive in the sense that conceptually any formally expressed property can be

verified to hold or not on a system model. However, as we will shortly explain, formal verification of hybrid systems is severely limited in practicality.

Focus now on the bouncing ball model. Assume that for a given set of initial conditions, i.e. initial height and velocity of the ball, we want to check if the ball comes to a halt within a specific point in time, t_1 . In other terms we want to check if $v = 0 \wedge h = 0 \wedge t \leq t_1$ holds. Ideally, for the specified property, a hybrid system verification tool performs an automated check on the model and returns a result in a reasonable time. Prominent hybrid system modelling and verification tools include Uppaal (Behrmann et al., 2006), Phaver (Frehse, 2008), and SpaceEx (Frehse et al., 2011). For a comparative analysis of hybrid system verification tools, please refer to (Carlioni et al., 2006).

Formal verification on hybrid systems are performed via *reachability analysis*. Reachability analysis is the process of computing the set of reachable states for a system model. In this context, a state of a hybrid system is a valuation of all the system variables. The formal verification problem can be encoded as a reachability analysis problem as follows. Given a hybrid system model M with a set of initial states I and a set of safe states S , which represents the property we want to prove or disprove on the model, we want to check every execution of M starting from an initial state in I always stays within the set of safe states S . In our bouncing ball example, M is the model in Figure 2.1a, I is the initial values for h and v , and S is the set of states conforming to $v = 0 \wedge h = 0 \wedge t \leq t_1$. Note that, implicit to the bouncing ball model, time is evolving with the differential equation $\dot{t} = 1$.

In general, a reachability algorithm, called a symbolic reachability algorithm as well, iteratively tries to compute the set R of reachable states of a given model M from the set I of initial states. Due to the complexity of computing a reachable set for a hybrid system, computation of R is usually an over-approximation of the set of actually reachable states. At every step, the algorithm checks whether R is contained in the set S of safe states. If it is the case, the algorithm continues, otherwise it returns with a counter-example which demonstrates the state evolution of the hybrid system that violates the property being verified. For an introduction on reachability analysis methods on hybrid systems, please refer to (Frehse, 2015).

As we laid out, numerical simulation and formal verification are two vastly different analysis methods. Numerical simulation is a widely accepted technique in industry due to the availability of precise and highly scalable algorithms. However, a single simulation run is dictated by the given initial conditions and the external stimuli provided during the run. Exhaustively simulating all the potential behaviours of a system for all foreseen initial conditions, parameters and external stimuli is in most cases impossible due to the large scale of the input domain. It is then left to the designer and helper tools to pick a relevant set of inputs for simulation. In any case, there is no guarantee that an interesting corner case is not missed. Another problem with numerical simulation is that simulation loses its effectiveness if the simulated system is non-deterministic. Non-determinism is a very powerful tool in modelling as it is a very natural way to model uncertainties and abstractions seen, for example, in early

stages of a design. In case of non-determinism, a simulation algorithm has to choose a path in order to continue with the simulation, increasing the set of possible runs for a particular initial condition.

Formal verification, at least conceptually, addresses these concerns of simulation as it can provide definitive results due to exhaustive reachability analysis performed on the state space of the model. However, formal verification of hybrid systems is severely limited due to two reasons, which have kept the technique from gaining widespread industrial application. First of all, for all but a very limited sort of hybrid systems—limited in the sense of allowed sort of differential equations and number of variables—reachability analysis is an undecidable problem. We will provide details on this part in Chapter 5. Second, even for the decidable segment of verification problems, reachability analysis on a hybrid system is not scalable for industrial use as the computation cost sharply increases with the number of continuous variables involved in the differential equations (Carloni et al., 2006; Frehse, 2015).

Verification by simulation or *symbolic simulation* is a promising effort to mix two methods applying advancements on symbolic reachability analysis algorithms to enhance numerical simulation techniques to cope with non-determinism and to increase coverage in simulations (Girard and Pappas, 2006; Alur et al., 2008). The idea is, for a given set of inputs, to use a single simulation trace of a system to compute a subset of inputs using symbolic analysis techniques which are proven to be close enough all the time. Then the chosen simulation trace can be used to show particular properties of the system that applies to the computed subset of inputs.

A variant of verification using simulation can be performed via *statistical model checking*. Given a stochastic system S and a property ϕ , the problem of statistical model checking is to check if S satisfies ϕ with a probability greater or equal to a certain threshold θ (Legay et al., 2010). In the case of a given hybrid system and a property to verify on it, the idea is to derive a statistical abstraction of a hybrid system by performing simulations on it. Then, statistical model checking can be applied on the resulting statistical model to check if the system satisfies the given property with a certain probability (Basu et al., 2010).

2.2 Contract-Based Design

In this section we present the details of the contract-based design framework that we base our specification and design efforts in Part II and Part III upon. Contract-based design is not a new idea. The idea first gained popularity in the design of transformational software systems. Bertrand Meyer pioneered the idea of *Design by Contract* for object-oriented design within the context of the Eiffel programming language (Meyer, 1992). The idea is to specify relationships between systems in terms of preconditions, postconditions and invariants. A contract specifies that an operation, that is started execution in a state and with inputs dictated by preconditions, conforms the invariants defined on the state and terminates ensuring the state and the outputs conform to the postconditions.

Such use of contracts specifies the behaviour of a sequentially acting components or

operations and are not suitable for specifying reactive components that operate concurrently, interacting with each other, as it is in embedded control systems. Our explanations in this section are based on a contract-based design framework by Benveniste et al., which aims to utilize contracts in the context of control systems (Benveniste et al., 2012).

2.2.1 A Meta-Theory of Contracts

As we mentioned in Section 1.1, the two concepts that make a complex design task manageable are abstraction/refinement and composition/decomposition. The very basic idea of contract-based design is to be able to utilize these concepts by specifying components in an abstract way such that analysis of composability and emergent behaviour of composition can be performed on the level of contracts and refinement of a contract by another contract or an implementation can be checked.

In (Benveniste et al., 2012), a mathematical meta-theory of contracts is presented. In an abstract way, the meta-theory defines the basic concept of a contract, i.e. what a contract captures, and the operations necessary to employ contracts in a component-based design methodology. This theory can be made concrete by specializing it using concrete formalisms to express the contracts and to support the required operations. Here we briefly mention the meta-theory and then we will present the instantiation of the theory by assume/guarantee contracts. Our explanations in this section are inspired by (Benveniste et al., 2012).

A *contract*, denoted by C , specifies a component, denoted by M or E , with the following two characteristics:

1. A contract is intentionally abstract.
2. A contract differentiates the component and its environment.

The *composition* operation, denoted by $M_1 \times M_2$, is defined between two components to compose them resulting in a new component. Composition is associative and commutative. Two components that can be composed are called *composable*. The components that can interact with the outside world are called *open*, and otherwise *closed*. An *environment*, denoted by E , for a component M is a component composable with M such that $M \times E$ is closed.

A contract is a pair $C = (E_C, M_C)$ where E_C is the set of environments of C and M_C is the set of implementations for C . A contract possessing no environment is called *incompatible* and a contract possessing no implementation is called *inconsistent*. For a given environment E , if $E \in E_C$ then E is an environment for C , which is denoted by

$$E \models^E C$$

Similarly, for a given component M , if $M \in M_C$ then M is an implementation for C , which is denoted by

$$M \models^M C$$

Between two contracts, the *refinement* operation is defined such that when C refines C' , denoted by $C \leq C'$, then any implementation of C also implements C' and any

environment for C' is also an environment for C :

$$C \leq C' \implies M_C \subseteq M_{C'} \wedge E_C \supseteq E_{C'}$$

The *conjunction* of two contracts formalizes the notion of having a set of contracts over a component. Conjunction of two contracts, denoted by $C = C_1 \wedge C_2$, is the greatest lower bound of C_1 and C_2 with respect to the refinement order. Conjunction of two contracts is computed as the contract formalizing the intersection of sets of implementations and union of sets of environments. As a result, any contract that refines $C_1 \wedge C_2$ also refines C_1 and C_2 . Any component that implements $C_1 \wedge C_2$ also implements C_1 and C_2 . Any component that is an environment for $C_1 \wedge C_2$ is also an environment for C_1 and C_2 .

Then the important concept of *contract composition* between two contracts is defined such that for C_1 and C_2 , the composition, denoted by $C_1 \otimes C_2$, ensures that:

1. Composition of any two implementations of C_1 and C_2 is an implementation of $C_1 \otimes C_2$:

$$M_1 \models^M C_1 \wedge M_2 \models^M C_2 \implies M_1 \times M_2 \models^M C_1 \otimes C_2$$

2. Composition of an environment for $C_1 \otimes C_2$ with an implementation of C_1 or C_2 yields an environment for C_2 or C_1 , respectively:

$$E_\otimes \models^E C_1 \otimes C_2 \wedge M_i \models^M C_i \implies E_\otimes \times M_i \models^E C_j \text{ for } (i, j) \in \{(1, 2), (2, 1)\}$$

Contract composition is very useful in supporting independent implementability. Let C_1 , C_2 , C'_1 and C'_2 be four contracts such that $C'_1 \leq C_1$, $C'_2 \leq C_2$, and C_1 and C_2 are compatible. Then, C_1 and C_2 are also compatible, and C_1 and C_2 can be independently refined/implemented by C'_1 and C'_2 :

$$C'_1 \otimes C'_2 \leq C_1 \otimes C_2$$

Contract composition is associative and commutative. For the lemmas and further corollaries on the meta-theory we refer the reader to (Benveniste et al., 2012).

2.2.2 Assume/Guarantee Contracts

In this subsection we describe assume/guarantee (A/G) contracts as an instance of the meta-theory of contracts. Inspired by (Benveniste et al., 2012), we will describe A/G contracts in terms of set of behaviours and employ set operations assuming a finite way of representing such behaviours supporting set operations. Later in Part II we will employ A/G contracts based on the particular definition of behaviours suitable for describing the behaviours of embedded control systems.

A/G contracts are expressed with *assertions*. An assertion is a set of behaviours defined over the ports/variables of the component. Let us assume all components and contracts share an alphabet Σ of variables of domain D . Then, an assertion P can be defined as:

$$P \subseteq \Sigma \mapsto D^* \cup D^\omega$$

where

The above definition expresses an assertion as a subset of all finite or infinite behaviours over alphabet Σ . According to this definition, a component M is identified with an assertion. Component composition is defined by intersection:

$$P_1 \times P_2 = P_1 \wedge P_2$$

An *A/G contract* is a pair $C = (A, G)$ of assertions called the assumptions and guarantees. The set of environments E_C of contract C is defined as the set of all assertions E satisfying $E \subseteq A$. Such an assertion E is called a legal environment for C . The set of implementations M_C of C is defined as the set of all assertions M satisfying $A \times M \subseteq G$. Such an assertion M is called a valid implementation of C . According to the definition, if a component, when put in a legal environment for C , satisfies the guarantees of C , then it is a valid implementation of C .

Note that for a given contract $C = (A, G)$, any component $M \subseteq G \cup \neg A$ is an implementation. The component $M_C = G \cup \neg A$ is called the maximal implementation of C . Any two contracts $C = (A, G)$ and $C' = (A', G')$ such that $G \cup \neg A = G' \cup \neg A'$ are equivalent. In other terms, any contract $C = (A, G)$ is equivalent to a *saturated contract* (A, G') such that $G' = G \cup \neg A$. A saturated contract is called consistent if $G \neq \emptyset$, and compatible if $A \neq \emptyset$.

The refinement relation is defined as follows:

$$C \leq C' \iff A \supseteq A' \wedge G \subseteq G'$$

Contract composition on A/G contracts is defined over the saturated contracts as follows. Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. The parallel composition $C = (A, G) = C_1 \otimes C_2$ is given by:

$$\begin{aligned} A &= (A_1 \cap A_2) \cup \neg(G_1 \cap G_2) \\ G &= G_1 \cap G_2 \end{aligned} \tag{2.1}$$

Note that the newly obtained composition contract is also saturated.

The intuition of parallel composition of A/G contracts is as follows (Benveniste et al., 2008). If $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ are contracts (possibly over different sets of ports), the composite $C = C_1 \otimes C_2$ must satisfy the guarantees of both, implying an operation of intersection on the guarantees, hence $G = G_1 \cap G_2$. For assumptions, if the two contracts have disjoint sets of ports, the assumptions of the composite should be simply the conjunction of the assumptions, since the environment should satisfy all the assumptions. On the other hand, in general, part of the assumptions of C_1 or C_2 will be already satisfied by composing them, acting as a partial environment. Therefore, guarantees of one contract can contribute to relaxing the assumptions of the other. The following equations formalize this intuition:

$$\begin{aligned} A \cap G_1 &\subseteq A_2 \\ A \cap G_2 &\subseteq A_1 \end{aligned}$$

As we stated in Subsection 2.2.1, the conjunction operation formalizes the intuitive notion of a 'set of contracts' or a 'set of requirements' defined over a contract. Conjunction of two contracts is defined as:

$$C_1 \wedge C_2 = (A_1 \vee A_2, G_1 \wedge G_2)$$

For saturated A/G contracts conjunction is indeed associative and commutative. Note that, as stated in Subsection 2.2.1, $C_1 \wedge C_2$ is the greatest lower bound of C_1 and C_2 with respect to the refinement relation:

$$C_1 \wedge C_2 \leq C_1$$

$$C_1 \wedge C_2 \leq C_2$$

2.2.3 System Obligations

In the previous section we defined the conditions for an A/G contract $C = (A, G)$ to be consistent, i.e. $G \neq \emptyset$, and compatible, i.e. $A \neq \emptyset$. However, even a consistent and compatible contract can specify an empty set of behaviours. To show this, let us compute, for a given saturated A/G contract $C = (A, G)$, the maximal behaviour of an implementation when it is put in a legal environment. According to the definition of valid implementations for C , i.e. $A \cap M \subseteq G$, the maximal implementation is $M = G \cup \neg A$. Similarly, according to the definition of legal environments, i.e. $E \subseteq A$, the maximal environment is $E = A$. Then the maximal behaviour we can observe from the composition of maximal implementation and maximal environment is:

$$E \cap M = A \cap G$$

Note that even with non-empty A and G , it is possible to have $A \cap G$ empty. Such a contract would be useless since it does not define any behaviours, i.e. its legal environments and implementations are inconsistent. This is because, according to our definition of A/G contracts, a contract cannot ensure that a component or its environment show a particular behaviour. In other terms, A/G contracts are useful in expressing safety properties but not liveness properties. This is indicated by that the refinement and implementation relations in the contract algebra are defined in terms of subset relation.

For this reason, to check the validity of contracts, we use the concept of *system obligations* (Benveniste et al., 2008) which is defined as follows:

$$B_C = A \cap G$$

System obligations are useful in the sense that for a given contract C , having non-empty system obligations implies that the contract specifies a non-empty set of behaviours. Furthermore, that a contract $C = (A, G)$ has non-trivial system obligations, $B_C = A \cap G \neq \emptyset$, implies that the contract is both compatible and consistent:

$$(B_C \neq \emptyset) \Rightarrow (A \neq \emptyset \wedge G \neq \emptyset)$$

Similarly, if a contract is inconsistent or incompatible, then it has empty system obligations:

$$(A = \emptyset \vee G = \emptyset) \Rightarrow (B_C = \emptyset)$$

A contract $C = (A, G)$ *conforms* to a system obligation B if $B_C \subseteq B$. A contract $C' = (A', G')$ *conforms* to a contract $C = (A, G)$ if C' preserves C 's system obligations, $B_{C'} \subseteq B_C$. This is denoted by $C' \leq_c C$. Note that conformance is compositional:

$$B_{C_1 \otimes C_2} = B_{C_1} \cap B_{C_2}$$

Figure 2.3: Venn diagram demonstration of assume and guarantee assertions of contracts C and C' , where $C' \leq_c C$

Making a contract saturated preserves its system obligations: $B_C = A \wedge G = A \wedge (G \vee \neg A) = B_{C'}$. Conformance and refinement between contracts are complementary, in the sense that one does not imply the other. To see that refinement does not preserve system obligations, let $C = (A, G \cup \neg A)$ be a consistent and compatible contract with assertions A and G depicted in Figure 2.3. Note that C has non-empty obligations, $B_C = A \cap G \neq \emptyset$. Also let $A' \supseteq A$ and $G' \subseteq G$ be two sets as shown in Figure 2.3. Then, we can define the following 3 variants of the saturated contract $C_i \leq C$ for $i \in [1, 3]$:

- $C_1 = (A', G' \cup \neg A')$
- $C_2 = (A', G \cup \neg A')$
- $C_3 = (A, G' \cup \neg A) = (A, \neg A)$

The obligations for $C_i, i \in [1, 3]$ have varying properties: $B_{C_1} \cap B_C = \emptyset$, $B_{C_2} \supseteq B_C$, $B_{C_3} = \emptyset$. If a contract C' both refines and conforms to a contract C , then it is called that C' *strongly refines* C . Strong refinement is denoted by $C' \leq_s C$:

$$C' \leq C \wedge C' \leq_c C \iff C' \leq_s C$$

That the refinement of a contract does not preserve system obligations is not a problem, since a contract specifies a contract's behaviour in an assumed set of environments. A refinement can relax assumptions, hence it can introduce new behaviour due to the newly allowed environment behaviours. However, it is a problem when a

refinement ends up in a contract with empty system obligations, i.e. inconsistent environments and implementations. Furthermore, even with a saturated A/G contract with non-empty set of system obligations, composition of a valid environment and a valid implementation might have an empty set of behaviours. Let $C = (A, G \cup \neg A)$ be a saturated contract such that $B_C = A \cap G \neq \emptyset$. Then, if $A \not\subseteq G$, it is possible for the composition of a valid environment E ($E \subseteq A$) and a valid implementation M ($M \subseteq G \cup \neg A$) to have an empty behaviour set: $E \cap M = \emptyset$. These sorts of inconsistencies between assumptions/guarantees of a refining contract or between individual environments/implementations for a contract can happen since, as we pointed out previously, an A/G contract does not enforce any behaviours on a component or on its environment, i.e. it is inherent to the contracts framework. In Part II and Part III, our methods of defining and validating contracts will provide mitigation for this issue.

2.2.4 Practical Implementation of Assume/Guarantee Contracts with Tests

The effective use of presented A/G contracts algebra relies upon the concrete formalisms to express assertions and the practical methods to manipulate them. As outlined in Subsection 2.2.2, performing contract operations such as conjunction and composition, and checking contract relations such as refinement, consistency, compatibility and implementation requires the ability to perform set operations on assertions. In the presence of behaviours with infinite data types and length, as it is the case in expressing behaviours of reactive components in embedded control systems, directly performing aforementioned set operations on assertions is not feasible. In Chapter 5 we will make this infeasibility clear for the sort of behaviours we use to specify embedded control systems when we introduce the concrete formalism to define them.

In order to make the A/G contracts framework effective with the assertions with infinite domains, employing *tests* is an option. Tests are used to create effective representations of contracts, tackling the infinite domains the contracts address (Benveniste et al., 2012).

Definition 2.2.1. A *test* is a deterministic program attached to a property or a requirement, taking all ports of the considered interface as inputs and delivering a boolean output. The boolean output is initially true and it switches to false and remains so as soon as the encoded property or requirement gets falsified. $\square$

The idea of using tests in A/G contracts is as follows. Let's assume that, for the domain of behaviours D with an alphabet Σ , we know a way of formulating any assertion as a test. Given an assertion $P \subseteq \Sigma \mapsto D^* \cup D^\omega$, T_P is a test formulating P such that for a given behaviour $\rho \in \Sigma \mapsto D^* \cup D^\omega$ of possibly infinite length:

$$T_P(\rho) \text{ outputs } F \implies \rho \notin P$$

Using tests we can express a contract $C = (A, G)$ as (T_A, T_G) . Performing contract composition and conjunction on contracts expressed with tests is straightforward as the tests are defined to be deterministic. Determinism enables us to combine tests using

Boolean operations, i.e. apply conjunction, disjunction and negation on the outputs of the tests.

Checking contract refinement, implementation of a contract by a model, and consistency and compatibility of a contract can only be partially performed using tests. This is because, as stated in Subsection 2.2.2 and Subsection 2.2.3, checking for the mentioned relations requires either checking for set inclusion (for refinement and implementation) or for emptiness (for consistency and compatibility). Checking set inclusion ($A \subseteq B$) or emptiness ($B = \emptyset$) requires exhaustively checking whether each behaviour in a set (A or U) is included in another set (B). Unfortunately, for this reason, it is not possible to perform a total verification of contract relations for infinite domains. In Part II, we will provide further guidance on how to employ tests to use A/G contracts effectively.

3

Case Study Description

In this chapter we describe a case study which we will use to demonstrate our approach in Part II and Part III. Our case is specification and design of embedded software for a maze navigation robot called R2-G2Px. In Section 3.1, we describe the hardware and kinematics of R2-G2Px. In Section 3.2, we describe the maze structure and the modes of operation of the robot to navigate the maze.

Figure 3.1: The R2-G2Px robot (Oguz et al., 2012)

3.1 The R2-G2Px Robot

In Figure 3.1, R2-G2Px mobile robot is shown (Oguz et al., 2012). R2-G2Px is a mobile, two-wheeled differential drive robot with 15cm length and 8cm width. R2-G2Px is equipped with an AVR board with an Atmega32 microcontroller and a Gumstix Overo FireSTORM board with a 1GHz ARM Cortex-A8 microprocessor. The operating system on the Gumstix is a Linux system. The AVR board and the Gumstix communicate via a serial connection. The Gumstix hosts all the computation whereas the AVR board serves as an interface between the Gumstix and the sensors and actuators. All the

sensors and actuators are connected to the pins of AVR board. The sensors and the actuators on the R2-G2Px are as follows:

- Four infra-red sensors for line following.
- Two infra-red distance sensors that sense objects in the forward driving direction.
- A contact switch in the front that senses physical contacts with an object.
- Two encoders (one per wheel) that sense the angular position of the wheels.
- Two servo motors that are connected to the wheels and driven by PWM (Pulse Width Modulation) signals.

Figure 3.2: The R2-G2Px robot on a referential frame.

R2-G2Px is a two-wheeled differential drive robot as depicted on a referential frame in Figure 3.2. In the figure v and ψ denote the robot's forward speed and the angular rotation rate. v and ψ can be represented in terms of functions of the left and right wheel speeds of the robot, w_l and w_r , the radius of the wheel, r (3cm), and half of the wheel base, l (4cm), as follows:

$$v = \frac{r(w_r + w_l)}{2}$$

$$\psi = \frac{r(w_r - w_l)}{2l}$$

Then the robot's kinematics can be described as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \psi \end{bmatrix}$$

where x , y , and θ denote robot's position and orientation with respect to the frame depicted in Figure 3.2. In the following chapters, we will denote robot's kinematics by

two shorthand equations:

$$\dot{p} = \text{Pos}(\theta, w_l, w_r)$$

$$\dot{\theta} = \text{Ang}(w_l, w_r)$$

where $p = (x, y)$ is the tuple for robot's position.

Figure 3.3: The R2-G2Px robot on a sample maze

3.2 Maze Navigation

R2-G2Px is a line follower robot. Our case study for the robot is navigating through a maze composed of paths defined by lines printed on the ground and obstacles placed at the borders of the maze. The maze is structured such that the robot can explore it following the edges and turning at the junctions that are printed on the ground.

A maze is defined by placing blocks on a regular grid of lines with edge length of 70cm. A piece of block can only be placed such that it is aligned with the grid lines and it starts and ends at the centers of grid cells. In this way an edge between two junctions is either unblocked or blocked right at the middle. Furthermore, the blocks should be placed such that the area inside of the maze is closed. A sample maze is shown in Figure 3.3.

The task for the robot is to visit all the navigable junctions in the maze using a exploration algorithm. Initially the robot is placed on the grid such that its center of rotation is on one of the junctions. The forward driving direction of the robot should be aligned with one of the four edges coming out of the junction. A sample initial location for the robot is shown in Figure 3.3.

The design goal is to enable the robot navigate the maze so that, when coupled with a traversal algorithm, the robot can explore the whole maze. We have defined 5 basic motions for the robot to perform on the described maze such that when a sequence of motions are performed in an appropriate order, then the robot can navigate a maze. Each basic motion is described by a mode of operation, the descriptions of the motions within the operational modes are as follows:

m_1 : The stop mode. Robot stays still on a junction or on a line.

- m_2 : The line-follow mode. Robot drives following the line.
- m_3 : The junction mode. For this mode the robot should be in line-follow mode on a junction. Then the robot positions itself on the exit of the junction.
- m_4 : The rotate mode. The robot should be in stop mode placed on the exit of a junction or on a line. Then the robot rotates, leaving the line it is aligned with, until it positions itself on the line it first encounters.
- m_5 : The back-up mode. For this mode the robot should be in line follow mode and in front of an obstacle. The the robot stops and backs up to clear itself from the object.

In Figure 3.4, the allowed transition between the modes of the robot are depicted in a state diagram. From the stop mode (m_1) the robot starts driving following the line (m_2) or starts rotating (m_4) until it positions itself again on the line. While driving on the line (m_2), the robot can go into junction mode (m_3) if it faces a junction or it can go into back-up mode (m_5) if it faces an obstacle. In the junction mode (m_3), the robot is supposed to stop (m_1) when it positions itself at the exit of the junction, aligned with the line. In the rotate mode (m_4), the robot is supposed to stop (m_1) when it aligns itself with a line after it leaves the line it was aligned with when it starts rotating. In the back-up mode (m_5), the robot is supposed to stop (m_1) when it clears itself from the obstacle, keeping its alignment with the line.

Figure 3.4: Transitions between the modes of operation of the R2-G2Px robot.

Part II

Specification

4

System Specification with Modes

In this chapter, we describe our approach of constructing a system specification with good quality. Our approach is to structure a system specification based on system modes and mode-switching logic and express the overall specification as an assume/guarantee (A/G) contract. We describe our approach independent of a particular formalism to express the specification contract.

In Section 4.1, we provide the definition of system specification. Then, we present the quality criteria for a good quality specification. In Section 4.2, we introduce the notion of ‘system modes’ which is used to structure a specification. Lastly, in Section 4.3, we present an architecture to construct a system specification with modes and mode-switching logic. We introduce a workflow to obtain an overall system specification contract structured into system modes from an initial set of textual requirements. We also provide the validation actions to be taken in the workflow to ensure the good quality of the resulting specification contract.

4.1 System Specification

In this section, first, we describe the notion of system specification and importance of it. We review two mainstream reference models for system specification. Then, we present a new model which we will base our specification approach upon. Last, we state the quality criteria used to assess the quality of specifications.

For computer systems, system specification or system requirements specification is the description of the behaviour of the computer system to be developed. For safety-critical embedded systems, completeness and consistency of system specification is vital to the safe and correct functioning of the end product.

In (Lutz, 1993), Lutz examined 387 software errors discovered during the integration and system testing of the Voyager and Galileo spacecraft. The result is that the safety-related software errors were most commonly caused by (1) “discrepancies between the documented requirements specifications and the requirements needed for correct functioning of the system” and (2) “misunderstandings about the software’s interface with the rest of the system”.

It is stated in (Leveson, 2000) that “Accidents and major losses involving computers are usually the result of incompleteness or other flaws in the software requirements, not coding errors.”, and in (Leveson, 2004) that “Almost all software-related aerospace accidents and accidents in other industries have been related to flawed requirements and misunderstanding about what the software should do.”

In Figure 4.1, a simple view of a computer system in interaction with the environment is shown. In this picture, the system acts as a solution to a problem defined in the environment. By a *problem* we mean a well defined set of user requirements as in “air humidity within the house should be between 40% and 50%” and not a difficulty as in “on certain months of the year air humidity within houses can reach disturbing levels”. A *solution* to a problem is the particular way of solving the problem. A problem can have different solutions; a particular solution is valid as long as it solves the given problem. For the above humidity problem, various valid solutions might adopt different ways of measuring and manipulating the air humidity.

As shown in Figure 4.1, developing/realizing a solution for a problem is called design, which is performed on the system itself. On the other hand, the problem for which the system is developed as a solution resides in the environment. Therefore, the specification of the problem should be performed on the environment side, on the entities of the environment (Jackson, 2005). In this context, specification is both the act and result of expressing a problem using a suitable formalism. It is this specification of the problem that defines a description of the behaviour of the system to be developed, hence the system specification. Note that, according to this definition of system specification, the system to be developed is viewed/treated as a black-box. This way the specification and design are separated and the specification focuses solely on the problem, which is defined on the environment variables.

Figure 4.1: System and Environment

4.1.1 A Model for System Specification

In a system specification, a system is defined as a black-box component with a boundary, ports and variables. The contents of a system specification are the identified system ports and variables and a set of relations defined on them. A generic classification of the specification contents that describes the relationships among the variables and relations is important to identify and structure the contents correctly.

The Four-variable Model

A widely accepted perspective on the relationships among the variables and relations in a system specification is described by Parnas and Madey (1995). In their work, Parnas and Madey define what they call *the four-variable model* for system requirements and specification. The four-variable model is composed of four sets of variables and five relations among these (Figure 4.2). There are four distinct collections of variables located at the corners of Figure 4.2: *mon* for monitored values, *con* for system-controlled values, *input* for values input to the software, and *output* for output values of software. The monitored and controlled variables (*mon* and *con*) exist in the environment outside the system and they are limited to those that the system can directly sense and control, respectively. The variables contained in the sets *input* and *output* correspond to input and output registers of the system.

Figure 4.2: The four-variable model. Adapted from (Thompson et al., 2000).

A variable in one of the sets *mon*, *con*, *input* and *output* has a value which can be recorded as a function of time. For a variable v , v^t , as Parnas and Madey use it, denotes the time function describing the values of v . $v^t(t)$ denotes the value of v at time t . Then, a valuation of a set of variables is described by a vector of time-functions. For instance, for the set *mon* which is composed of n variables, $mon = \{m_1, m_2, \dots, m_n\}$, $(m_1^t, m_2^t, \dots, m_n^t)$ is a vector of time functions of *mon* containing one element for each of variables in *mon*. A system specification is then defined by five sets of binary relations defined between the sets of variables. A binary relation $B(V, Z)$ defined between two sets of variables $V = \{v_1, v_2, \dots, v_n\}$ and $Z = \{z_1, z_2, \dots, z_p\}$, is a set of pairs of vectors of time functions, where a pair is of sort $((v_1^t, v_2^t, \dots, v_n^t), (z_1^t, z_2^t, \dots, z_p^t))$. In Figure 4.2 the five relations defined between the sets of variables are shown. $NAT(mon, con)$ describes the environment (Parnas and Madey define the environment as the nature and the previously installed systems, hence the term NAT) which constrains the values of environmental quantities and $REQ(mon, con)$ describes the desired system behaviour further constraining the environmental quantities.

$IN(mon, input)$ relates the monitored environmental values to their correspond-

ing software inputs and $OUT(output, con)$ relates the software-generated outputs to system-controlled environmental values. $SOF(input, output)$ relates software inputs to software outputs.

In order to assess the validity of the specification, three predicates are defined on the relations in the four-variable model. First one is called *requirements feasibility* which ensures that the nature (NAT) should allow the required behaviour (REQ). The second ensures that the system should be *input enabled*: IN must be able to handle all the cases possible under NAT. Third one is called *software acceptability* which ensures that the software behaviour (SOF) should satisfy the requirements (REQ) when it is combined with IN, OUT and NAT relations.

In the four-variable model, the relations IN, OUT and SOF are used to express the software specification that satisfies the system specification. The system specification is essentially defined by NAT and REQ that collectively captures the system's assumptions on the environment and its promises as the effects on the environment.

One problem is that NAT and REQ need to be defined on the monitored and controlled variables, i.e. *mon* and *con*, which should be directly accessible by the system. However, initial user requirements might be defined on the environment variables that are not directly accessible to the system. Only after some design steps are performed, the requirements can be defined on the variables that are directly accessible to the system. For most systems, especially the ones that necessitate extensive control design, the focus should be on the entities on which the actual requirements are defined, as long as possible. The consistency and completeness should be analysed on these entities. The relations between the environment entities that the user is actually interested in and the entities directly accessible (visible and/or controllable) to the developed system are relatively unimportant and unrelated in the system specification process and they better belong to the design process. For instance, the same set of requirements can be met with different control designs which use different sets of sensors/actuators and/or different control algorithms which, in turn, would imply using different monitored and controlled variables. And such design steps should be taken after the initial requirements are analysed for consistency and completeness criteria, which are vital for embedded systems.

The WRSPM Reference Model

Another reference model for requirements and specification is proposed by Gunter et al. (2000). *The WRSPM reference model*, as the authors call it, is similar to the four-variable model and supplies what we miss in the four-variable model, i.e. the variables that are not directly accessible by the system. Within WRSPM reference model, there are five artefacts depicted in the top Venn diagram in Figure 4.3 that are used to define the system requirements and specifications:

W: World: The domain knowledge that describes the environment or nature.

R: Requirements: The system requirements that are described as the effects of the system on the environment.

S: Specification: The system specification that satisfies the requirements.

Figure 4.3: The WRSPM reference model. Adapted from (Gunter et al., 2000)

P: Program: The program that implements the system specification on the execution platform.

M: Machine: The execution platform that hosts the program.

There are also distinct sets of variables using which the above artefacts are defined. These are depicted at the bottom Venn diagram in Figure 4.3. The set of variables that belong to the environment and can be controlled by it is represented by e . The set of variables that belong to the system and can be controlled by it is represented by s . The variables in e that are visible to the system and that are hidden from it are represented by e_h and e_v , respectively. Similarly, the variables in s that are visible to the environment and that are hidden from it are represented by s_h and s_v , respectively. Controllability and visibility of the terms are indicated on the bottom of Figure 4.3.

The sets *input* and *output* in context of the four-variable model belong to s_h . The monitored and controlled variables of four-variable model, *mon* and *con*, are the same as e_v and s_v , respectively. There is no corresponding set in four-variable model for the set e_h ; actually this is the set of variables that we find lacking in four-variable model. The set e , in total, contains all the environmental variables that the user is actually interested in and the initial user requirements might be defined on.

The five artefacts in the WRSPM reference model are assumed to be described using higher-order logic using different sets of variables. The artefacts $W(e_v, e_h, s_v)$ and $R(e_v, e_h, s_v)$ are defined on the sets of variables that are visible to the environment, which are e_v , e_h and s_v . The artefacts $P(s_h, s_v, e_v)$ and $M(s_h, s_v, e_v)$ are defined on the sets variables that are visible to the system, which are s_h , s_v and e_v . The specification artefact $S(e_v, s_v)$ are defined on the sets of variables that are visible to both the environment and the system, namely on the sets s_v and e_v .

The WRSPM reference model's main contribution is the specification relation S which acts like a contract between the “user” and the “supplier”. The user is responsible for drawing out the system requirements on the environment side by defining W and R

and obtaining S from these. The supplier is responsible for obtaining an implementation on the system side by defining P and M that implements S . Gunter et al. provide formulas on the W , R , S , P and M artefacts that the user and the supplier has to satisfy in order to ensure consistency and adequacy of their parts.

As previously mentioned, in contrast to the four-variable model, the WRSPM reference model covers all the environmental variables (the sets e and s_v), including the ones that are not directly accessible to the system. The system requirements, namely the artefacts W and R , are defined on these variables. However, we find the WRSPM reference model lacking necessary validity criteria regarding the system requirements. There is only one formula, namely the formula $\exists e s. W$ for consistency of the domain knowledge, among the proposed consistency and adequacy formulas that address consistency of the requirements.

Our Specification Model

Based on our observations on the presented models for specification and our need to express the specification with contracts, we define a new specification model. Our motivations for defining a new specification model are as follows:

1. Our approach is contract based. The aim is to express the system specification as an assume/guarantee contract.
2. As we explained in the discussion about the four-variable model, we argue that environment variables should not be classified as controlled/uncontrolled by the system or as visible/hidden to the system. One exception to the above item is the previously installed systems in the environment. From the perspective of the system to be developed, all the ports in the interface to the previously installed systems are visible and either controlled (inputs) or uncontrolled (outputs).
3. Unlike it is done in the four-variable model and in the WRSPM reference model, we do not define software specification and relationship between the system and software specifications as this is a part of the design process.

In Figure 4.4, the boundaries of the entities in our model and the ports and variables involved are shown. Note that the system boundary, depicted by the dashed box, covers the software and also a part of the environment indicating that the system has its effects on the environment. Figure 4.5 depicts a refined version of Figure 4.4 where the environment is decomposed into a controlled plant and the rest of the environment (shown as primed environment).

The ports and variables in Figure 4.4 and Figure 4.5 are as follows:

- O_{os} : Outputs to other systems. Discrete, controllable.
- I_{os} : Inputs from other systems. Discrete, uncontrollable.
- O_e : Outputs to environment. Discrete, controllable.
- I_e : Inputs from environment. Discrete, uncontrollable.
- V_s : System variables. Discrete. Superset of all the ports: $V_s \supseteq I_{os} \cup O_{os} \cup I_e \cup O_e$.
- $V_e = V_p \cup V'_e$: Environment variables of interest: the variables of the controlled plant and the variables in the rest of the environment acting on the plant. Con-

tinuous, can be system-controlled or environment-controlled and visible or hidden to the system.

Figure 4.4: System ports and variables

Figure 4.5: System ports and variables with controlled plant and the rest of the environment separated

A system specification is defined on the variables and ports contained in $I_{os} \cup O_{os} \cup V_e$. Regarding the environment-side variables, we do not make distinction between the variables that are controlled/uncontrolled by and/or visible/hidden to the system. The aim is to formulate the specification on the actual entities that the user requirements are naturally defined on, which are included in V_e . On the other hand, the ports on the interface to the other systems, O_{os} and I_{os} , are all visible to the system to be developed and classified as controlled or uncontrolled by the system.

The specification consists of two sets, A_{sys} and G_{sys} , of the histories of the variables contained in $I_{os} \cup O_{os} \cup V_e$. The set $A_{sys}(I_{os}, O_{os}, V_e)$ defines the assumptions made on the environment, whereas the set $G_{sys}(I_{os}, O_{os}, V_e)$ defines the promises of the system. As we describe in Section 4.3, our system specification, composed of the A_{sys} and G_{sys} sets, is formulated as an assume/guarantee contract, $C_{sys} = (A_{sys}, G_{sys})$.

4.1.2 Quality Criteria for Requirements Specification

In this subsection, we explain the importance of quality criteria for requirements specifications. We mention the criteria used in the literature to evaluate the quality of requirements specifications.

Most of the accidents involving computers are due to incompleteness and inconsistent requirements specifications (Lutz, 1993; Leveson, 2000, 2004). Therefore, it is crucial not only to formulate the system specification but also to assess and ensure the quality of it. Unless the correctness of the requirements specifications ensured, the end product will contain errors however successful the implementation is.

In literature, a variety of characteristics of specifications with ‘good’ quality are identified (Wieringa, 1996; Zowghi and Gervasi, 2003; Haskins, 2012). Here, we provide high-level definitions for such characteristics that are independent of the way or method chosen to formulate the specifications.

In (Zowghi and Gervasi, 2003), *correctness*, *completeness* and *consistency*, termed as 3C’s, are identified as the three main properties of good quality requirements. There are different definitions and uses of these concepts in the literature (See (Zowghi and Gervasi, 2003) for a detailed discussion). Our interpretations of these concepts, which we believe are the most suitable regarding system specification, are as follows:

Consistency A specification is consistent if all of the parts of it are consistent with each other. In other words, a specification is consistent if it does not involve any conflicts within. In this respect, consistency is an internal property of the specification. The particular formalization of the consistency property depends on the formalism that the specification is expressed in.

At the stage of requirements specification, what we mean by consistency is logical consistency, i.e. being free of contradictions. A related notion is realizability which defines whether there can actually be a system that realizes the specification. An example of a consistent but unrealisable specification is the one that defines a behaviour for which a system has to foresee the future to realize. For this reason, consistency at the specification stage does not necessarily imply realizability. Consistency in later design stages has a stronger relation to realizability. On the other hand, inconsistency of a requirements specification implies unrealizability.

Completeness A specification is complete if it specifies all the pertinent parts of the solution that is specified. In this respect, completeness is an external property of the specification since it refers to an external entity, the solution. The problem is that, this entity is not defined completely elsewhere. Therefore, it is not possible to define and measure absolute completeness. Instead, achieving *sufficient completeness* should be the aim. The typical nature of control systems and hazard analysis allow to define general guidelines for assessment of completeness. For instance, one such guideline dictates that start-up and shut-down behaviour of a safety-critical system should be completely defined. In (Leveson, 2000), it is mentioned that they identified a set of 60 formal completeness criteria “to identify missing, incorrect, and ambiguous requirements for process-control systems”, which are applicable to any behavioural requirements specification described with a state machine-based formalism. These criteria can be used to assess the completeness of a system specification.

Correctness Correctness can only be defined with respect to another entity. Thus, like completeness, correctness is an external property. Correctness of a top-level sys-

tem specification is defined with respect to the user goals. A top-level system specification is correct if the specification meets all the user goals, as much as possible. In this respect, correctness is property of a top-level specification that is sufficient to make it a good specification. For top-level specifications, correctness is, indeed, implied by completeness and consistency. For a program or a non-top-level specification, correctness is defined with respect to the parent specification. A program is correct if it implements its specification. A non-top-level specification is correct if it refines its parent specification(s).

The three concepts defined above are used to define and assess quality of system specifications from the behavioural or functional point of view. There are other quality perspectives that can be taken. In the INCOSE System Engineering Handbook (Haskins, 2012) the characteristics of good requirements are defined including nonbehavioural ones such as affordable, clear, concise, bounded, necessary, etc. Nonbehavioural perspectives are important as well, however, in our work, we focus only on behavioural aspects.

4.2 Modes

In the previous section, we presented the contents of a system specification, namely, two sets of relations defined on the identified system variables. However, it was not defined how to group and express these relations, in order, for instance, to tackle complexity. In this section, first, we present the notion of *mode* which is useful in structuring the specifications. Then, we introduce our approach of specifying systems with explicit modes and a mode-switching logic, behaviours of which are specified using contracts.

In expressing the system specification, whatever the chosen formalism, there is a need for a structure in order to tackle complexity. The aim of this structure is to decompose the specification into logical parts without reflecting any design decisions. In the context of control systems, modes are found to be useful for grouping the relations of the specification into a set of main system functions (Maraninchi and Remond, 1998; Miller, 1998; Parnas, 2010). In general, system modes are defined according to identified main system functions which have specific set of objectives on the environment or the controlled plant. Then a mode-switching logic is used to supervise switching, i.e. activating and deactivating, of these functions according the system's overall objective.

Modes can be seen in all of the complex systems. For a flight guidance system they might correspond to flight control laws which are used to generate guidance commands. For a car's cruise control system, they might correspond to driving modes such as manual driving and cruise control. A system's modes can be mutually exclusive, i.e. only one mode is active at a given time. Mutually exclusive modes are composed sequentially. Alternatively, parallel modes can be allowed where parallel composition of modes is allowed as well. For instance, a car's driving modes, i.e. manual drive and cruise control, can be composed with one of its fuel saving modes, i.e. economy or performance.

A formal definition of a system mode can be given in terms of the system states (Parnas, 2010). System states are defined such that two states of a system are equal if the system's future behaviour is indistinguishable when the system has been at one of these states. The idea is that the states/histories in a mode form a meaningful set due to some logical decomposition of the system's behaviours such as a decomposition with respect to main system functions (Parnas, 2010). A system with a number of continuous variables would have infinite numbers of states. On the other hand, the number of modes of a system is usually finite.

In literature there are different interpretations of the presented definition of modes. In (Parnas, 2010), Parnas uses the notion of mode classes to strictly partition the states of the system. All mode classes are orthogonal. System state can belong to at most one mode in a mode class and all modes in a class are mutually exclusive. A system-wide mode is then defined as an n -tuple of all mode classes.

Modecharts (Jahanian and Mok, 1994) and Real-time Mode-Machines (Paynter, 1996) are statechart like formalisms which explicitly dwell on the notion of modes. In these works, authors choose to define modes as nonexclusive sets of states. Maraninchi and Remond (1998) also discuss the notion of mode, but as exclusive sets of states. The view of modes in these works are represented in the semantics of the proposed formalisms to express the system specification.

Our intuition on exhaustiveness is that the combination of all the modes of a system does not need to be exhaustive with respect to the whole system state space. The reason is that, in a system specification, mostly there are some assumptions on the environment. It is unnecessary to cover the state space regions that are in conflict with these assumptions. For instance, if there is an assumption declaring that two particular environmental events cannot occur within a certain proximity to each other, then there is no need to cover this possibility in the specification. Exhaustiveness with respect to the state space region that conforms to the environment assumptions should, on the other hand, be ensured for the sake of completeness of the specification. Regarding the exclusivity, the modes of a system should be exclusive as this would prevent confusion between the nonexclusivity due to the intentionally included non-determinism and errors.

Whatever such details of formal definition of modes in terms of system states are, the most important benefit of using modes is methodological which is that modes form meaningful states together according to main system functions. Therefore, in our approach, we focus on the methodological value of employing system modes and only require that a system mode is a set of system states without imposing any requirements on exhaustiveness or exclusivity.

Representation of mode-switching logic and the mode behaviour are important factors affecting expressivity and ease of use. In literature, to express mode-switching logic, formal specification languages are used, whether they are state-machine based formalisms (Leveson et al., 1994; Harel et al., 1990; Jahanian and Mok, 1994; Paynter, 1996; Maraninchi and Remond, 1998; The Mathworks Inc., 2016b) or tabular representations (Parnas, 2010; Faulk et al., 1992). Expressivity of these languages are usually

equal. Important factors for the selection of the formalism are clear representation and clear semantics.

Regarding expressing system's behaviour when it is at a particular mode, the tools Statecharts (Harel et al., 1990), Mode-automata (Maraninchi and Remond, 1998), Stateflow (The Mathworks Inc., 2016b) use dataflow languages. In these tools modes of the system are associated with equations or tasks defined on the inputs and outputs, which define the mode behaviour. These tools are primarily design oriented languages so they are not really aimed towards specification of a system. They lack the ways to specify behaviours in an abstract way.

The formalisms specifically aiming system specification (Leveson et al., 1994; Parnas, 2010), on the other hand, do not prescribe explicitly any formalism or method to specify the mode behaviours of the system. They focus on specifying just the mode-switching logic. They do not explicitly specify behaviours of individual modes, nor they look at the combination of modes with each other and with the switching logic.

As mentioned in the previous section, our approach to system specification is contract-based. We employ contracts to specify the mode-switching logic and mode behaviour of the system. For this, we regard modes and the mode-switching logic as components (of the specification not the design) and specify their behaviour with contracts.

The concrete formalism to express contracts is not fixed and can be chosen with respect to the nature of the behaviour that is to be expressed. We will provide details on this in the following chapters. However, the major benefits of using contracts are there whatever the chosen concrete formalism. First, by using contracts we focus on the black-box behaviour of the components, i.e. of the mode-switching and mode-composition components, which is essential for the specification process. Second, using contracts, we are able to make explicit distinction between the environment and the component itself. This is important for ensuring the consistency in the combination of the modes with each other and with the switching logic.

In the next section, we give details of our contract-based specification approach based on a mode-based architecture. Our contributions within this approach will be twofold: Technical contributions, such as the formalisms used for contracts, and methodological contributions such as the details of specification process employing the contracts and modes.

4.3 A Specification Architecture with Modes

In this section, we present our specification architecture which comprises modes and mode-switching logic: First, we describe the initial system decomposition into mode and mode-switching components. Then, we give the formulation of the system contract as composition of contracts of its mode and mode-switching components. Then, we define a workflow to construct a system specification contract from the initial system requirements (Subsection 4.3.1). Last, we present the quality criteria used to assess the quality of a contract-based specification (Subsection 4.3.2). We also state the benefits of our specification approach on the quality of the system specification.

Our approach is to define an architecture for system specification where modes are represented by components along with a component responsible for mode-switching behaviour. We define each mode as a component, and we specify the components with A/G contracts. Then, the abstract contract theory presented in Subsection 2.2.1 is applicable for defining the system specification.

The initial system decomposition is shown in Figure 4.6. The *mode-switching* component denotes the component that is responsible for the mode-switching logic, i.e. activating and deactivating the modes of the system. The *mode-composition* component denotes the combination of the modes of the system.

Note that, as it is the case with the overall system specification, specification of the mode-composition component does not refer to the the ports I_e and O_e but the environment variables V_e . The communication between the mode-switching component and the mode-composition component might include more than the events and commands for mode switching. The mode-switching component might provide parameters to the modes and modes can report back values relevant to the particular mode's objective.

According to this architecture, the overall system specification contract, C_{sys} , can be formulated as the composition of the contracts of the mode-switching and mode-composition components (See Section 2.2 for contract composition):

$$C_{sys} = C_{sw} \otimes C_{mc} \quad (4.1)$$

where C_{mc} and C_{sw} are mode-composition and mode-switching contracts, respectively.

In Figure 4.7, an example for the further decomposition of the system is shown where the mode-composition component is decomposed into three mode components. Each mode component has a direct communication with mode-switching component and environment. However, mode components do not communicate directly with each other even though more than one mode can be active at the same time. This is a decision to promote defining system modes that are fairly independent of each other. If there are two modes that are active at the same time and if those require direct communication with each other to achieve their particular objectives, then this suggests that definition of the system modes should be revised.

The contract formulation for mode-composition component is then the composition of all the contracts for all the mode components:

$$C_{mc} = C_{m_1} \otimes C_{m_2} \otimes C_{m_3} \quad (4.2)$$

Figure 4.6: Initial system decomposition.

Figure 4.7: Mode-composition component is decomposed into a number of modes.

4.3.1 Workflow for Specification

In this subsection, we define a workflow (Figure 4.8) to obtain a system specification from an initial set of informal system requirements. We briefly explain each step and outcome in the process, which will be explained in detail in the following chapters. The aims of this process are:

1. Starting from a set of initial system requirements, R_{sys} , construct an overall system specification, C_{sys} , in the form of assume-guarantee contracts conforming to the mode-based architecture proposed in the previous section.
2. During the construction process, check and ensure the validity of the specification with respect to the quality criteria.

As seen in Figure 4.8, at the beginning of the process, we assume a list of informal requirements, R_{sys} , describing a number of operational modes and switching logic. The requirements refer to the system ports and variables. Event though we favour a specification architecture with system modes, it is out of our scope to provide guidelines on how to define modes of a system. Instead, we focus on constructing and validating a system specification with modes using contracts. For this aim, we assume a set of requirements structured into modes as a starting point.

In the workflow, the first step is *requirements decomposition*, in which R_{sys} is decomposed into separate sets of requirements for mode switching and individual modes:

$$(R_{m_1} \otimes \dots \otimes R_{m_n} \otimes R_{sw}) \leq R_{sys}$$

Even though we assume that R_{sys} describes modes and switching logic, we do not assume initially it is composed of separate sets of requirements for mode switching and individual modes. For instance, there might be an item in R_{sys} that mentions more

Figure 4.8: System specification workflow.

than one mode. Specification of switching and modes using contracts assumes separate sets of requirements for mode switching and the individual modes. These sets should refer to the interface between the modes and switching. Also, as seen in Figure 4.7, mode-switching component interacts with the plant only through the modes; its specification does not refer to the plant variables. These are achieved in the *requirements decomposition* step by modifying the requirements and obtaining separate sets of requirements for switching and the individual modes. We might also need to add some requirements and some existing requirements might be split. Note, however, that this altering process is complex and can be error prone. The errors, however, as we will see, can be detected during the validation steps of the workflow performed on the contracts.

In the step for *formalization and validation of mode contracts* the contracts for individual modes of the system, C_{m_i} , are constructed from the mode requirements, R_{m_i} , such that:

$$\forall i \in [1, n] \ C_{m_i} \leq R_{m_i}$$

Also individual mode contracts are validated with respect to the quality criteria we describe in Subsection 4.3.2. In the case of any errors discovered during the validation, the mode contracts which are the sources of the error are revised. Then, in the step for *composition of the mode contracts*, the mode contracts are composed into the mode-composition contract, C_{mc} :

$$C_{mc} = C_{m_1} \otimes \dots \otimes C_{m_n}$$

There is no validation involved after C_{mc} as its validity is implied by the validity of the individual mode contracts (Section 6.2). The two steps for constructing the mode contracts and then the mode-composition contract are explained in detail in Chapter 6.

In the step for *formalization and validation of the mode-switching contract*, the mode-switching contract C_{sw} is constructed from the mode-switching requirements such that:

$$C_{sw} \leq R_{sw}$$

Then C_{sw} is validated with respect to the validity criteria we describe in Subsection 4.3.2. In the case of any errors discovered during the validation, the mode-switching contract is revised. This step is explained in detail in Chapter 7.

In the last step, *composition of the mode and mode-switching contracts* is performed to obtain the overall system specification contract C_{sys} by composing C_{mc} and C_{sw} :

$$C_{sys} = C_{mc} \otimes C_{sw}$$

At this time, C_{mc} and C_{sw} , the constituents of the overall system contract C_{sys} , are already validated. Building on this information, in validation of C_{sys} , we only need to ensure the validity of the system behaviour while the system switches between the modes or it has more than one active modes, as dictated by the mode-switching specification. This step is explained in detail in Chapter 8.

4.3.2 Validation of Contract-Based Specifications

In Subsection 4.1.2, we introduced three criteria to assess the quality of specifications. In this subsection, we apply these criteria for contract-based specifications that appear in the presented workflow. We also state the benefits of our specification approach with mode-based architecture and contracts on the quality of the system specification.

In (Benveniste et al., 2012), four essential quality criteria specific for contract-based specifications are presented. These criteria are the same as the ones we presented in Subsection 4.1.2, which are correctness, completeness and consistency, with the addition of a compatibility criterion. Here, compatibility represents the consistency of the assumptions of the contract on the environment whereas the consistency criterion is limited to refer to the internal consistency of the component implementation specified by the contract. We take the definitions in Benveniste (Benveniste et al., 2012) as basis. However, in order to be compatible with the requirements specification literature, we do not use the compatibility criteria. Instead we use consistency criteria to refer the consistency of the contract regarding both assumptions on the environment and the guarantees provided by component implementation.

In the following parts we explain how the three quality criteria are defined and used for the sets of requirements and contracts in the workflow.

Correctness

For the top level system requirements R_{sys} , correctness is implied by consistency and completeness (see Subsection 4.1.2). So we do not assess correctness of R_{sys} directly. In the *requirements decomposition* step, correctness of the separate sets requirements $R_{m_1}, \dots, R_{m_n}, R_{sw}$ are defined with respect to R_{sys} . We call the requirements $R_{m_1}, \dots, R_{m_n}, R_{sw}$ correct if they collectively refine R_{sys} :

$$(R_{m_1} \otimes \dots \otimes R_{m_n} \otimes R_{sw}) \leq R_{sys}$$

Similarly, correctness of the contracts in the workflow implies that they refine the preceding contract(s) or requirements. For instance, the contract C_{m_k} for the k th mode is correct if it refines the sets of requirements R_{m_k} for that mode:

$$C_{m_k} \leq R_{m_k}$$

The switching contract C_{sw} is correct if:

$$C_{sw} \leq R_{sw}$$

The overall system contract C_{sys} and the mode-composition contract C_{mc} are direct formulations in the form of contract compositions so, by definition, they are correct with respect to their constituent contracts.

Completeness

A set of requirements R or a contract C is called *complete* if R or C defines all the necessary properties of the entity that is specified by R or C . For instance, the set of

requirements R_{m_k} or the contract C_{m_k} for the k^{th} mode is complete if R_{m_k} or C_{m_k} specifies all the pertinent parts of the behaviour expected from that mode. Completeness of a set of requirements R or a contract C is denoted by $\kappa(R)$ or $\kappa(C)$.

As stated in Subsection 4.1.2, for R or C , the external reference for completeness, that is the specified entity, is not fully defined. Also, the only truly complete specification of an entity would be the entity itself. For these reasons, it is impossible to attach a definition to $\kappa(R)$ or $\kappa(C)$ for absolute completeness. Instead $\kappa(R)$ or $\kappa(C)$ denotes *sufficient completeness* of a set of requirements R or a contract C (Zowghi and Gervasi, 2003).

For the textual requirements $R_{\text{sys}}, R_{m_1} \dots R_{m_n}, R_{\text{sw}}$, we do not assess the completeness directly. For the contracts in the workflow, completeness is assessed by model executions as it will be explained in chapters 6, 7 and 8. Then, the completeness of the initial requirements R_{sys} is either validated or, fixed, which is usually the case since initial textual requirements are rarely complete.

Consistency

In (Benveniste et al., 2012) compatibility and consistency of a contract are defined as the feasibility of the legal environments and component implementations, respectively. In order to be compatible with the requirements specification literature (Zowghi and Gervasi, 2003), we leave out the compatibility criteria for a contract and call a contract consistent if both the legal environments and implementations described by the contract are feasible. In the validation steps all the contract-based specifications are assessed for consistency. Definitions and assessing methods for consistency of the contract-based specifications for modes, mode switching and overall system are given in chapters 6, 7 and 8, respectively.

As opposed to the contracts within our workflow, which are assume-guarantee contracts, in the sets of textual requirements $R_{\text{sys}}, R_{m_1}, \dots, R_{m_n}, R_{\text{sw}}$ there is no sharp distinction between environment and component yet. Again consistency of an individual set of requirements defines the feasibility of the environment as well as the component. However, as it is with the completeness, we do not assess the consistency of the textual requirements directly. Their consistency, is either validated or fixed, indirectly, by the consistency assessment performed on the contracts.

4.4 Conclusions

As stated in Subsection 4.1.2 ensuring quality of a system specification is essential. For a specification defined for a set of requirements, our approach with *A/G contracts and specification architecture with modes* enables and enhances the process of assessing and improving the three quality criteria, i.e. correctness, completeness and consistency. The specification architecture with modes naturally helps with achieving completeness since it is a suitable way for structuring a control system specification. For instance, the completeness of C_{sys} builds on top of the completeness of C_{mc} and C_{sw} . Also, with A/G contracts, separation of component's responsibilities from the ones of environment helps to assess completeness of specification contract.

The approach for system specification presented in this chapter is independent of any particular formalism to express contracts. In the next chapter, we will present a concrete formalism suitable for expressing A/G contracts for system specification. Then, in Chapters 6, 7 and 8, we will explain how to employ this formalism to construct a mode-based specification contract. There, we will see more concretely how the ideas on system specification presented in this chapter, i.e. employing a mode-based architecture and A/G contracts, enhances the quality of a system specification.

5

A/G Contracts for System Specification

In this chapter, we define the concrete formalism for assume/guarantee (A/G) contracts used to specify components in the system specification phase. The abstract A/G contracts framework is introduced in Section 2.2. However, the abstract framework does not set a concrete formalism to express behaviours of a component. Here, we state particular formalisms to express the behaviours of a component and assertions. Our main aim is to define a concrete contract formalism that is expressive enough to specify control systems, formal and exact to avoid ambiguities, and, at the same time, practical to use in industry-adopted tools.

An A/G contract for a component is a pair of assertions, A and G , which represent the assumptions and guarantees of the component. In Sections 5.1 and 5.2, based on the literature, we present the formal definitions for hybrid assertions, assertions capable of expressing hybrid behaviour. In Section 5.1, we define assertions on the component ports/variables as subsets of all hybrid trajectories over the component ports/-variables. In Section 5.2, we present the hybrid automata formalism which is used to express hybrid assertions as finite representations. Then, based on the hybrid automata, we define open hybrid automata, an extension of hybrid automata formalism with input and output variables, and composition operation.

In Sections 5.3 and 5.4, we explain how to enable the A/G contract framework presented in Section 2.2 for hybrid contracts in theory and in practice. In Section 5.3, we explain how to form hybrid observers based on open hybrid automata formalism, which we use to express assertions that can be complemented and that are online-testable. Complementability is required to apply the contract algebra presented in Section 2.2 and online-testability is required to use the observer as a test while working with hybrid system simulation tools. The sufficient conditions for a hybrid observer to be complementable and online-testable are determinism and nonblockingness. One contribution of this chapter is, inspired by the conditions for a regular hybrid automaton to be deterministic and nonblocking, we define the formalism of open hybrid automata with urgent transitions to arrive at testable hybrid observers.

In Section 5.4, we explain the practical methods for using hybrid observers in hybrid system simulation tools. Another contribution of this chapter is that we describe how

to construct a hybrid observer within 20-sim, and methods to check if a component model conforms to the assertion dictated by the observer using simulation. Lastly, in Section 5.5, we present how the quality criteria introduced in Section 4.1 and Section 4.3 translate into the concrete A/G contracts formalism presented in this chapter.

5.1 Assertions

In order to specify the behaviour of control systems, which are digital computers embedded in an analog environment, we need to be able to describe behaviours of both digital and continuous nature. In this section we give definitions of such hybrid behaviours and assertions to express sets of hybrid behaviours.

Behaviours of a component are defined over the ports/variables of that component. For a component M , we represent all the ports of it by its alphabet P . Each port/-variable $p \in P$ has a domain of values denoted by $D(p)$. A port can be discrete or continuous valued.

We express hybrid behaviours of a component with hybrid trajectories defined over the ports of that component. A *hybrid trajectory* contains both continuous intervals, where continuous evolution take place, and discrete points in time, where discrete transitions happen. The sequence of intervals of a hybrid trajectory is called its *hybrid time set*, definition of which is given in Definition 5.1.1 (Lygeros et al., 2008).

Definition 5.1.1. A *hybrid time set* is a finite or infinite sequence of intervals $\tau = \{I_0, I_1, \dots, I_N\} = \{I_i\}_{i=0}^N$, such that

- $I_i = [\tau_i, \tau'_i]$ for all $i < N$ where $\tau_i, \tau'_i \in \mathbb{R}^+ \cup \{0\}$;
- if $N < \infty$ then either $I_N = [\tau_N, \tau'_N]$ or $I_N = [\tau_N, \tau'_N)$ where $\tau_N, \tau'_N \in \mathbb{R}^+ \cup \{0, +\infty\}$;
- $\tau_i \leq \tau'_i$ for all $i \leq N$ and $\tau'_i = \tau_{i+1}$ for all $i < N$.

□

Example 5.1.1. The following is an example hybrid time set:

$$\tau = \{[10, 15], [15, 15], [15, 27], [27, 28)\}$$

□

The set of all hybrid time sets is denoted by $\boldsymbol{\tau}$. The *discrete extent* of a hybrid time set, denoted by $\langle \tau \rangle$, is defined as a function $\boldsymbol{\tau} \rightarrow \mathbb{N} \cup \{\infty\}$:

$$\langle \tau \rangle = \begin{cases} N & \text{if } \tau \text{ is a finite sequence} \\ \infty & \text{if } \tau \text{ is an infinite sequence} \end{cases}$$

The discrete extent of the hybrid time set in Example 5.1.1 is 3, $\langle \tau \rangle = 3$. The *continuous extent* of a hybrid time set, denoted by $\|\tau\|$, reflects the length of the time period that the hybrid time set spans:

$$\|\tau\| = \sum_{i=0}^N (\tau'_i - \tau_i)$$

Note that $\|\tau\|$ is a finite summation when $N < \infty$ and an infinite series when $N = \infty$. The continuous extent of the hybrid time set in Example 5.1.1 is 18, $\|\tau\| = 18$.

We classify the hybrid time sets as follows:

Definition 5.1.2. A hybrid time set τ is called:

- **Finite**, if τ is a finite sequence and last interval of τ is closed.
- **Finite-open**, if τ is a finite sequence and last interval of τ is bounded and right-open.
- **Infinite**, if τ is an infinite sequence or the sum of the time intervals in τ is infinite, i.e. $\langle \tau \rangle = \infty$ or $\|\tau\| = \infty$.
- **Admissible**, if τ is an infinite sequence and the sum of the time intervals in τ is infinite, i.e. $\langle \tau \rangle = \infty$ and $\|\tau\| = \infty$.
- **Zeno**, if τ is an infinite sequence but the sum of intervals in τ is finite, i.e. $\langle \tau \rangle = \infty$ and $\|\tau\| < \infty$.

□

Example 5.1.2. The hybrid time set in Example 5.1.1 is finite-open. The following three hybrid time sets are all infinite. Also, $\hat{\tau}$ is admissible and $\bar{\tau}$ is zeno.

$$\begin{aligned}\tau &= \{[0, 1], [1, 2], [2, 3], [3, \infty)\} \\ \hat{\tau} &= \{I_i\}_{i=0}^{\infty}, I_i = (i, i+1) \\ \bar{\tau} &= \{I_i\}_{i=0}^{\infty}, I_i = \left(\frac{i}{i+1}, \frac{i+1}{i+2}\right)\end{aligned}$$

□

A hybrid time set $\tau = \{I_i\}_{i=0}^N$ is called a *prefix* of $\hat{\tau} = \{\hat{I}_i\}_{i=0}^K$ if either $\tau = \hat{\tau}$ or τ is finite, $N \leq K$, $I_i = \hat{I}_i$ for $i \in [0, N-1]$ and $I_N \subseteq \hat{I}_K$. We denote this prefix relation by $\tau \sqsubseteq \hat{\tau}$. If a hybrid time set τ is a prefix of a hybrid time set $\hat{\tau}$ such that $\tau \neq \hat{\tau}$, then we call τ a *strict prefix* of $\hat{\tau}$, which is denoted by $\tau \sqsubset \hat{\tau}$.

Definition 5.1.3. For two hybrid time sets $\bar{\tau} = \{\bar{I}_i\}_{i=0}^M$ and $\tau = \{I_i\}_{i=0}^N$ and with $M < \infty$ and $\bar{\tau}'_M = \tau_0$ we define two combination operations:

$comb_s(\tau, \bar{\tau}) = \hat{\tau}$ where

$$\hat{\tau} = \{\hat{I}_i\}_{i=0}^{M+N} \text{ and } \hat{I}_i = \begin{cases} \bar{I}_i & \text{if } i \in [0, M-1] \\ [\bar{\tau}_M, \tau'_0] & \text{if } i = M \\ I_{i-M} & \text{if } i \in [M+1, M+N] \end{cases}$$

$comb_p(\tau, \bar{\tau}) = \hat{\tau}$ where

$$\hat{\tau} = \{\hat{I}_i\}_{i=0}^{M+N+1} \text{ and } \hat{I}_i = \begin{cases} \bar{I}_i & \text{if } i \in [0, M] \\ I_{i-M-1} & \text{if } i \in [M+1, M+N+1] \end{cases}$$

□

In Definition 5.1.3, the $comb_s$ operation joins the last interval of the first operand with the first interval of the second whereas the $comb_p$ keeps them separate. Using the combination operations we can define postfix relation for hybrid time sets. A hybrid time set $\tau = \{I_i\}_{i=0}^N$ is called a *postfix* of $\hat{\tau} = \{\hat{I}_i\}_{i=0}^K$ if one of the following two conditions hold

1. $\tau = \hat{\tau}$
2. There is a prefix $\bar{\tau} = \{\bar{I}_i\}_{i=0}^M$ of $\hat{\tau}$ such that when combined with τ gives $\hat{\tau}$, i.e. either $comb_p(\bar{\tau}, \tau) = \hat{\tau}$ or $comb_s(\bar{\tau}, \tau) = \hat{\tau}$.

We denote this postfix relation by $\tau \dot{\supseteq} \hat{\tau}$. If a hybrid time set τ is a postfix of a hybrid time set $\hat{\tau}$ such that $\tau \neq \hat{\tau}$, then we call τ is a *strict postfix* of $\hat{\tau}$, which is denoted by $\tau \dot{\subset} \hat{\tau}$.

For a set S of variables, a valuation is a mapping where each variable $s \in S$ is mapped to a value in $D(s)$. Let $\mathbf{S}$ denote the set of all valuations for a set S of variables. A *hybrid trajectory* is defined as in Definition 5.1.4

Definition 5.1.4. A *hybrid trajectory* over a collection of variables W is a pair (τ, w) consisting of a hybrid time set $\tau = \{I_i\}_0^N$ and a sequence of functions $w = \{w_i\}_0^N$ with $w_i : I_i \rightarrow \mathbf{W}$. □

Example 5.1.3. Over a set of variables $W = \{d, x\}$ with domains $D(d) = \{0, 1\}$ and $D(x) = \mathbb{R}$, an example hybrid trajectory (τ, w) can be defined, where $\tau = \{I_i\}_0^N$ is the hybrid time set in Example 5.1.1 and $w = \{w_i\}_0^N$ is defined as follows:

$$w_i(t) = (d_i(t), x_i(t)) = \begin{cases} (0, t) & \text{if } i \text{ is even} \\ (1, t) & \text{if } i \text{ is odd} \end{cases}$$

□

The classifications defined in Definition 5.1.2 naturally extend to hybrid trajectories. So the hybrid trajectory in Example 5.1.3 is finite-open. A hybrid trajectory (τ, w) over W is called a *prefix* of another hybrid trajectory $(\hat{\tau}, \hat{w})$ over W if $\tau \sqsubseteq \hat{\tau}$, and for all $i \in \langle \tau \rangle$ and all $t \in I_i$, $w_i(t) = \hat{w}_i(t)$. We denote this prefix relation by $(\tau, w) \sqsubseteq (\hat{\tau}, \hat{w})$. A hybrid trajectory (τ, w) is a *strict prefix* of a hybrid trajectory $(\hat{\tau}, \hat{w})$, denoted by $(\tau, w) \sqsubset (\hat{\tau}, \hat{w})$, if $(\tau, w) \sqsubseteq (\hat{\tau}, \hat{w})$ and $(\tau, w) \neq (\hat{\tau}, \hat{w})$. Within a set of hybrid trajectories, a hybrid trajectory is called *maximal* if it is not a strict prefix of any other hybrid trajectory in the same set.

Definition 5.1.5. For two hybrid trajectories $(\bar{\tau}, \bar{w}) = (\{\bar{I}_i\}_{i=0}^M, \{\bar{w}_i\}_{i=0}^M)$ and $(\tau, w) =$

$(\{I_i\}_{i=0}^N, \{w_i\}_{i=0}^N)$ and with $M < \infty$ and $\bar{\tau}'_M = \tau_0$ we define two combination operations:

$comb_s((\bar{\tau}, \bar{w}), (\tau, w)) = (comb_s(\bar{\tau}, \tau), \{\hat{w}_i(t)\}_{i=0}^K)$ where

$$\hat{w}_i(t) = \begin{cases} \bar{w}_i(t) & \text{if } i \in [0, M-1] \\ \bar{w}_M(t) & \text{if } i = M \text{ and } t < \bar{\tau}'_M \\ w_0(t) & \text{if } i = M \text{ and } t \geq \bar{\tau}'_M \\ w_{i-M}(t) & \text{if } i \in [M+1, K] \end{cases}$$

$comb_p((\bar{\tau}, \bar{w}), (\tau, w)) = (comb_p(\bar{\tau}, \tau), \{\hat{w}_i(t)\}_{i=0}^K)$ where

$$\hat{w}_i(t) = \begin{cases} \bar{w}_i(t) & \text{if } i \in [0, M] \\ w_{i-M-1}(t) & \text{if } i \in [M+1, K] \end{cases}$$

□

In Definition 5.1.5, the $comb_s$ operation joins the last interval of the first operand with the first interval of the second whereas the $comb_p$ keeps them separate. Using the combination operations we can define postfix relation for hybrid trajectories. We can also define a *postfix* relation between two hybrid trajectories. A hybrid trajectory (τ, w) over W is called a *postfix* of another hybrid trajectory $(\hat{\tau}, \hat{w})$ over W if one of the two following conditions hold:

1. $(\tau, w) = (\hat{\tau}, \hat{w})$
2. There is a prefix $(\bar{\tau}, \bar{w})$ of $(\hat{\tau}, \hat{w})$ such that when combined with (τ, w) gives $(\hat{\tau}, \hat{w})$, i.e. either $comb_s((\bar{\tau}, \bar{w}), (\tau, w)) = (\hat{\tau}, \hat{w})$ or $comb_p((\bar{\tau}, \bar{w}), (\tau, w)) = (\hat{\tau}, \hat{w})$.

We denote this postfix relation by $\tau \dot{\sqsubseteq} \hat{\tau}$. A hybrid trajectory (τ, w) is a *strict postfix* of a hybrid trajectory $(\hat{\tau}, \hat{w})$, denoted by $(\tau, w) \dot{\sqsubset} (\hat{\tau}, \hat{w})$, if $(\tau, w) \dot{\sqsubseteq} (\hat{\tau}, \hat{w})$ and $(\tau, w) \neq (\hat{\tau}, \hat{w})$.

The set of all the hybrid trajectories over W , denoted by $Traj(W)$, is defined as follows:

$$Traj(W) = \{(\tau, w) \mid \tau \in \mathbf{T} \wedge \tau_0 = 0 \wedge w = \{w_i\}_0^{(\tau)} \text{ where } w_i \in [I_i \rightarrow \mathbf{W}]\}$$

The set of all infinite trajectories over W , denoted by $bTraj^\infty(W)$, is defined as follows:

$$bTraj^\infty(W) = \{(\tau, w) \mid \tau \in Traj(W) \wedge (\langle \tau \rangle = \infty \vee \|\tau\| = \infty)\}$$

The *join(h)* operation on a hybrid trajectory $h = (\tau, w)$ joins all the adjacent intervals in τ where $w_i(\tau'_i) = w_{i+1}(\tau_{i+1})$ (note that $\tau'_i = \tau_{i+1}$):

Definition 5.1.6. Given a hybrid trajectory $h = (\tau, w)$, *join(h)* is a hybrid trajectory resulting from joining all the adjacent intervals in h repetitively where $w_i(\tau'_i) = w_{i+1}(\tau_{i+1})$:

$$join((\tau, w)) = \begin{cases} (\tau, w) & \text{if } \langle \tau \rangle \leq 1 \\ comb_s((\tau_0, w_0), join((\{\tau_i\}_{i=1}^{(\tau)}, \{w_i\}_{i=1}^{(\tau)}))) & \text{if } w_0(\tau'_0) = w_1(\tau_1) \\ comb_p((\tau_0, w_0), join((\{\tau_i\}_{i=1}^{(\tau)}, \{w_i\}_{i=1}^{(\tau)}))) & \text{otherwise} \end{cases}$$

□

Note that for two hybrid trajectories h_1 and h_2 with $h_1 \neq h_2$ it is possible that $join(h_1) = join(h_2)$. The *projection* of a hybrid trajectory h over a set of variables W onto a set of variables $W' \subseteq W$ allows us to eliminate some variables in W from h :

Definition 5.1.7. Given a hybrid trajectory $h = (\tau, w)$ over a set W , the *projection* of h onto a set $W' \subseteq W$ is denoted by $pr_{W'}(h)$. The projection of h onto W' is defined as follows:

$$pr_{W'}(h) = (\tau, w') \text{ where} \\ w' = \{w'_i\}_0^{(\tau)} \text{ and for all } v \in W', w'_i(t)(v) = w_i(t)(v) \text{ for } t \in [\tau_i, \tau'_i]$$

□

We use hybrid trajectories to express the behaviours of components. A behaviour of a component M with alphabet P is expressed by a hybrid trajectory over P . All the possible behaviours of a component M with alphabet P are then defined by $Traj(P)$ which is the set of all the possible hybrid trajectories over the variables in P .

In order to specify behaviours of components we use the notion of *assertion*. We are only interested in the infinite behaviours of a component since we would like to specify reactive systems that never blocks/deadlocks. An assertion L for a component M with alphabet P is a subset of all the infinite behaviours of M , i.e. $L \subseteq Traj^\infty(P)$. In this respect, our definition of assertions is analogous to the linear time properties of transition systems (Baier and Katoen, 2008).

On the assertions defined over the same alphabet, the binary set operations union, intersection and complement can be applied to form new assertions. The projection operation we defined for hybrid trajectories in Definition 5.1.7 applies to assertions. The *projection* operation is used to project an assertion defined over a particular alphabet onto a different alphabet. For an assertion L over an alphabet P , we define projection $pr_{P'}(L)$ for $P' \subseteq P$ to project L onto P' . $pr_{P'}(L)$ is simply the set of all the trajectories contained in L projected to P' . The inverse projection of L , $pr_{P''}^{-1}(L)$, for $P'' \supseteq P$ is defined as the maximal subset of all the trajectories over P'' such that, when projected onto P , is equal to L if all trajectories of either side are joined:

$$join(pr_P(pr_{P''}^{-1}(L))) = join(L)$$

Using the projection and inverse projection we can define a handy operator for *alphabet equalization*. For an assertion L with an alphabet P_L , alphabet equalization is used to project L to an alphabet P which might be not a subset or a superset of P_L . Alphabet equalization is denoted by $pr_P^{\sim}(L)$ and defined as:

$$pr_P^{\sim}(L) = join(pr_P^{-1}(pr_{P \cap P_L}(L)))$$

5.2 Hybrid Automata

An assertion L over an alphabet P is usually an infinite set. However, we would like to express L in a compact way. This is possible using hybrid automata. A hybrid automaton (Henzinger, 1996) is a formal model capturing the discrete and continuous

Figure 5.1: An example for hybrid automata

behaviours of systems. It is a finite-state automaton equipped with continuous variables. In a hybrid automaton, discrete transitions are represented by the edges of the automaton, whereas continuous evolutions of the continuous variables are expressed with differential equations embedded in the locations of the automaton. Below, we provide the syntax of hybrid automata by (Lygeros et al., 2008).

Definition 5.2.1. A *hybrid automaton* H is a collection $H = (Q, X, Init, f, Inv, E, G, R)$, where

- Q is a finite set of discrete variables, $\mathbf{Q}$ is finite and called the set of locations;
- X is a finite set of continuous variables;
- $Init: \mathbf{Q} \rightarrow P(\mathbf{X}) := 2^{\mathbf{X}}$ assigns to each $q \in \mathbf{Q}$ an initial set;
- $f: \mathbf{Q} \times \mathbf{X} \rightarrow \mathbf{X}$ is a vector field where value of the f at each point of $\mathbf{Q} \times \mathbf{X}$ is a vector quantity of dimension $|X|$;
- $Inv: \mathbf{Q} \rightarrow P(\mathbf{X})$ assigns to each $q \in \mathbf{Q}$ an invariant set;
- $E \subset \mathbf{Q} \times \mathbf{Q}$ is a collection of discrete transitions;
- $G: E \rightarrow P(\mathbf{X})$ assigns to each $e = (q, q') \in E$ a guard; and
- $R: E \times \mathbf{X} \rightarrow P(\mathbf{X})$ assigns to each $e = (q, q') \in E$ and $x \in \mathbf{X}$ a reset relation.

□

Example 5.2.1. In Figure 5.1, a simple hybrid automaton is shown. Locations are drawn as ellipses and edges as arrows. There are two locations, $\mathbf{Q} = \{t_1, t_2\}$. A predicate next to a location denotes an invariant predicate defining $Inv(q)$. Invariants equivalent to *true* are omitted. A predicate in a box and next to a location is an initial predicate defining $Init(q)$. Initial predicates equivalent to *false* are omitted. An initial location, i.e. a location with a *non-false* predicate, is identified with a short arrow pointing to it, which is the t_1 location in Figure 5.1. Predicates inside the locations express flow predicates, i.e. $f(q, \cdot)$. Dotted variables, e.g. $\dot{x}$ denote first derivatives. Edges are embedded with a guard predicate and a reset predicate. Primed variables, e.g. x' , in the reset predicates denote the updated variables.

In Figure 5.1, the only initial location is t_1 . There is a single variable x which is initially equal to zero. When the automaton is at t_1 , the value of x increases with a rate of 1 according to the flow predicate $\dot{x} = 1$ until $x \leq 20$ is false, which is the invariant for t_1 . When x exceeds 20, the automaton takes the top edge; the automaton goes to the t_2 location and x is reset with its own value due to the jump predicate $x' = x$. At t_2 , x

constantly decreases with rate -0.5 according to the flow predicate $\dot{x} = -0.5$ until the invariant $x \geq 0$ is false. When x gets below 0, the automaton takes the bottom edge; the automaton goes back to t_1 and x is reset with its own value due to the jump invariant $x' = x$. $\square$

As given in Definition 5.2.2, executions of hybrid automata are given in terms of hybrid trajectories.

Definition 5.2.2. We can define a hybrid trajectory of the a hybrid automaton H as (τ, w) over $W = Q \cup X$. We would like to use q and x as the sequences of functions, $q = \{q_i\}_0^N$ and $x = \{x_i\}_0^N$, denoting parts of $w = \{w_i\}_0^N$ for sets of discrete and continuous variables, Q and X , such that $w_i(t) = q_i(t) \cup x_i(t)$. Whenever convenient, we will denote such a hybrid trajectory as (τ, q, x) . Recall that, the intervals of a hybrid time set $\tau = \{I_i\}_{i=0}^N$ are denoted as $I_i = [\tau_i, \tau'_i]$. Then, an *execution of a hybrid automaton H* is a hybrid trajectory (τ, q, x) over $Q \cup X$, which satisfies:

1. Initial condition: $x_0(0) \in \text{Init}(q_0(0))$.
2. Discrete evolution: for all i ,
 - a) $e_i = (q_i(\tau'_i), q_{i+1}(\tau'_{i+1})) \in E$,
 - b) $x_i(\tau'_i) \in G(e_i)$,
 - c) $x_{i+1}(\tau_{i+1}) \in R(e_i, x_i(\tau'_i))$.
3. Continuous evolution: for all i with $\tau_i < \tau'_i$:
 - a) $q_i : I_i \rightarrow Q$ is constant over $t \in I_i$, i.e. $q_i(t) = q_i(\tau_i)$ for all $t \in I_i$;
 - b) $x_i : I_i \rightarrow X$ is the solution to the differential equation

$$\frac{dx_i}{dt} = f(q_i(t), x_i(t))$$

over I_i starting at $x_i(\tau_i)$; and,

- c) for all $t \in [\tau_i, \tau'_i]$, $x_i(t) \in \text{Inv}(q_i(t))$.

$\square$

A hybrid trajectory (τ, q, x) is an execution of a hybrid automaton H if it conforms to the restrictions listed in Definition 5.2.2. Restriction 1 dictates that the execution should start from an initial state for H . We can assume $\tau_0 = 0$ without loss of generality. Restrictions 2.a, 2.b and 2.c deal with the discrete transitions. 2.a dictates that there should be an edge $e \in E$ starting from $q_i(\tau'_i)$ and ending in $q_{i+1}(\tau'_{i+1})$. 2.b dictates that the continuous state, $x_i(\tau'_i)$, before the transition should be enabled by the guard $G(e)$. 2.c dictates that the continuous state, $x_{i+1}(\tau_{i+1})$, after the transition should conform to the reset map $R(e, x_i(\tau'_i))$. Restrictions 3.a, 3.b and 3.c deal with the continuous evolutions. 3.a dictates that along the continuous evolution discrete state should stay constant. 3.b dictates that the continuous evolution in I_i should be the solution for the differential equation specified by f for I_i which depends on the discrete state. 3.c dictates that the continuous state in I_i should be included in the invariant defined by the discrete state in I_i . The set of executions of a hybrid automaton H is denoted by $\text{Exec}(H)$. For a hybrid automaton H , an execution in $\text{Exec}(H)$ is *maximal* if it is not a strict prefix of any other execution in $\text{Exec}(H)$.

A *state* of a hybrid automaton H is a tuple $(q, x) \in \mathbf{Q} \times \mathbf{X}$ where q and x are valuations of the discrete and continuous variables. For a hybrid automaton H we define the reachable states as follows:

Definition 5.2.3. For a hybrid automaton H , a state $(\hat{q}, \hat{x}) \in \mathbf{Q} \times \mathbf{X}$ is *reachable* if there exists a finite execution (τ, q, x) of H such that $(q(\tau'_{\langle \tau \rangle}), x(\tau'_{\langle \tau \rangle})) = (\hat{q}, \hat{x})$. $\square$

The set of states reachable by H is denoted by $Reach(H) \subseteq \mathbf{Q} \times \mathbf{X}$. The set of reachable states $Reach(H)$ is also called the *state space* of H .

Using hybrid automata, we can express assertions for components. The set of executions $Exec(H)$ of a hybrid automaton $H = (Q, X, Init, f, Inv, E, G, R)$ defines an assertion over an alphabet $P_H = Q \cup X$. Also, using the alphabet equalization operator we introduced, we can project the executions of H to any other alphabet. We define the assertion defined by a hybrid automaton H over an alphabet P as $Asrt_P(H) = Pr_P^{\sim}(Exec(H))$. Note that a hybrid automaton may contain maximal, finite executions which we call *blocking executions*. For a hybrid automaton to define a valid assertion, which is composed of only infinite trajectories, we require that the hybrid automaton does not have any blocking executions. We will address non-blocking hybrid automata in Subsection 5.3.1.

5.2.1 Open Hybrid Automata

Definition 5.2.1 provides a definition for hybrid automata with no input or output variables. With such a hybrid automata formalism, every assertion need to be modelled in a monolithic way which might be impractical. In this subsection we provide the definition of open hybrid automata and composition of open hybrid automata. An open hybrid automaton is a hybrid automaton with input and output variables which can be composed with other open hybrid automata. Composition of hybrid automata opens up the possibility of defining large systems by composing smaller pieces. We first give the definition for open hybrid automata (Lygeros et al., 2008):

Definition 5.2.4. An *open hybrid automaton OHA* is a tuple $(Q, X, V, Y, Init, f, h, Inv, E, G, R)$, where:

- Q is a finite set of internal discrete state variables, $\mathbf{Q}$ is finite and called the set of locations;
- X is a finite set of internal continuous state variables;
- V is a finite set of input variables. We assume $V = V_D \cup V_C$, where V_D contains discrete and V_C contains continuous variables.
- Y is a finite set of output variables. We assume $Y = Y_D \cup Y_C$, where Y_D contains discrete and Y_C contains continuous variables.
- $Init: \mathbf{Q} \rightarrow P(\mathbf{X}) := 2^{\mathbf{X}}$ assigns to each location $q \in \mathbf{Q}$ an initial set;
- $f: \mathbf{Q} \times \mathbf{X} \times \mathbf{V} \rightarrow \mathbf{X}$ is a vector field;
- $h: \mathbf{Q} \times \mathbf{X} \rightarrow \mathbf{Y}$ is a vector field;
- $Inv: \mathbf{Q} \rightarrow P(\mathbf{X} \times \mathbf{V})$ assigns to each $q \in \mathbf{Q}$ an invariant set;
- $E \subset \mathbf{Q} \times \mathbf{Q}$ is a collection of discrete transitions;
- $G: E \rightarrow P(\mathbf{X} \times \mathbf{V})$ assigns to each $e = (q, q') \in E$ a guard; and

- $R : E \times \mathbf{X} \times \mathbf{V} \rightarrow P(\mathbf{X})$ assigns to each $e = (q, q') \in E$, $x \in \mathbf{X}$ and $v \in \mathbf{V}$ a reset relation.

□

In addition to definition of hybrid automata(Definition 5.2.1), an open hybrid automaton has inputs and outputs, denoted by V and Y . Values of the variables in Y are defined by the output function h which depends on the discrete (Q) and continuous state (X) of the automaton. The input variables in V are used in the definitions of the flow f , invariant Inv , guard G and reset R functions.

Executions of an open hybrid automaton are defined as follows:

Definition 5.2.5. An execution of an open hybrid automaton OHA is a hybrid trajectory $\chi = (\tau, q, x, v, y)$ over the set of variables $Q \cup X \cup V \cup Y$ satisfying:

1. Initial condition: $x_0(0) \in \text{Init}(q_0(0))$.
2. Discrete evolution: for all i , one of the following two conditions hold:
 - a) $(q_i(\tau'_i), x_i(\tau'_i)) = (q_{i+1}(\tau_{i+1}), x_{i+1}(\tau_{i+1}))$
 - b) $e_i = (q_i(\tau'_i), q_{i+1}(\tau_{i+1})) \in E$,
 $(x_i(\tau'_i), v_i(\tau'_i)) \in G(e_i)$, and
 $x_{i+1}(\tau_{i+1}) \in R(e_i, x_i(\tau'_i), v_i(\tau'_i))$.
3. Continuous evolution: for all i with $\tau_i < \tau'_i$:
 - a) $q_i : I_i \rightarrow Q$ is constant over $t \in I_i$, i.e. $q_i(t) = q_i(\tau_i)$ for all $t \in I_i$;
 - b) $x_i : I_i \rightarrow X$ is the solution to the differential equation

$$\frac{dx_i}{dt} = f(q_i(t), x_i(t), v_i(t))$$

over I_i starting at $x_i(\tau_i)$; and,

- c) for all $t \in [\tau_i, \tau'_i)$, $(x_i(t), v_i(t)) \in \text{Inv}(q_i(t))$.

4. Output evolution: for all i and for all $t \in \tau_i, y_i(t) = h_i(q_i(t), x_i(t))$

□

Note that, according to the Restriction 2 in Definition 5.2.5 on discrete transitions, there are two types of transitions possible:

- As defined in 2.a, an environmental transition can occur, taking place outside of the open hybrid automaton and affecting only the input variables.
- As defined in 2.b, an internal transition can occur, where the state changes.

Of an execution $\chi = (\tau, q, x, v, y)$ of an open hybrid automaton, only the part (τ, v, y) is visible to the outside world. A transition is called an internal transition if after the transition the values of the input and output variables stay unchanged: $(v(\tau'_i), y(\tau'_i)) = (v(\tau_{i+1}), y(\tau_{i+1}))$. In an execution, internal transitions can be eliminated by concatenating the trajectories either side of the transition using the *join* operation we explained in Definition 5.1.6. A *trace* of an open hybrid automaton is an execution restricted to the input and output variables, with all internal transitions removed and the corresponding trajectories concatenated:

Definition 5.2.6. A trace of an open hybrid automaton OHA is a hybrid trajectory $\chi = (\tau, v, y)$ over the set of input and output variables of OHA satisfying:

$$\chi \in \text{join}(pr_{V \cup Y}(\text{Exec}(OHA)))$$

□

$\text{Trace}(OHA)$ denotes the set of all traces of H . Two open hybrid automaton OHA_1 and OHA_2 are *comparable* if $V_1 = V_2$ and $Y_1 = Y_2$. Then, OHA_1 is said to *implement* OHA_2 if they are comparable, and $\text{Trace}(OHA_1) \subseteq \text{Trace}(OHA_2)$.

Composition of Open Hybrid Automata

Now we are ready to provide a definition for composition of hybrid automata. Two hybrid automaton can be composed with each other if they are *compatible*:

Definition 5.2.7. Two open hybrid automata OHA_1 and OHA_2 are *compatible* if the following three conditions hold:

1. $(Q_1 \cup X_1) \cap (Q_2 \cup X_2 \cup V_2 \cup Y_2) = \emptyset$
2. $(Q_2 \cup X_2) \cap (Q_1 \cup X_1 \cup V_1 \cup Y_1) = \emptyset$
3. $Y_1 \cap Y_2 = \emptyset$

□

The first and second items in Definition 5.2.7 state that internal variables $Q \cup X$ of one automaton should be disjoint from all the variables $Q \cup X \cup V \cup Y$ of the other. This is required since internal variables of an automaton are hidden from outside. The third item states that the output variables of the two automata should be disjoint. This is required since an output variable cannot be controlled by both automata.

For two compatible hybrid automata, OHA_1 and OHA_2 we define the following partitions of V_1 and V_2 to simplify the definition of composition:

$$\begin{aligned} V_1 &= V_{11} \cup V_{12} \text{ with } V_{11} = V_1 \setminus Y_2 \text{ and } V_{12} = V_1 \cap Y_2 \\ V_2 &= V_{21} \cup V_{22} \text{ with } V_{22} = V_2 \setminus Y_1 \text{ and } V_{21} = V_2 \cap Y_1 \end{aligned}$$

Let, for a collection of variables A , and a subset $A' \subseteq A$, a *restriction* of a valuation $a \in \mathbf{A}$, $a|_{A'}$, to be a valuation of the variables in A' that agrees with a . Then, for the partitions V_{12} and V_{21} , we define the maps:

$$\begin{aligned} h_{12} : \mathbf{Q}_1 \times \mathbf{X}_1 &\rightarrow \mathbf{V}_{21} \\ (q_1, x_1) &\mapsto h_1(q_1, x_1)|_{V_{21}} \\ &\text{and} \\ h_{21} : \mathbf{Q}_2 \times \mathbf{X}_2 &\rightarrow \mathbf{V}_{12} \\ (q_2, x_2) &\mapsto h_2(q_2, x_2)|_{V_{12}} \end{aligned}$$

Definition 5.2.8. The *composition* of two open hybrid automata OHA_1 and OHA_2 is a hybrid automaton $OHA = OHA_1 \parallel OHA_2 = (Q, X, V, Y, \Sigma, Init, f, h, Inv, E, G, R)$, with

- $Q = Q_1 \cup Q_2$;
- $X = X_1 \cup X_2$;
- $V = (V_1 \cup V_2) \setminus (Y_1 \cup Y_2)$;
- $Y = Y_1 \cup Y_2$;
- $Init = \{(q, x) \in \mathbf{Q} \times \mathbf{X} : (q|_{Q_i}, x|_{X_i}) \in Init_i, \text{ for } i=1,2\}$;
- $f : \mathbf{Q} \times \mathbf{X} \times \mathbf{V} \rightarrow \mathbb{R}^{n_1+n_2}$ given by:

$$f(q, x, v) = \begin{bmatrix} f_1(q|_{Q_1}, x|_{X_1}, v|_{V_{11}}, h_{21}(q|_{Q_2}, x|_{X_2})) \\ f_2(q|_{Q_2}, x|_{X_2}, v|_{V_{22}}, h_{12}(q|_{Q_1}, x|_{X_1})) \end{bmatrix}$$

- $h : \mathbf{Q} \times \mathbf{X} \rightarrow \mathbf{Y}$ is given by:

$$h(q, x) = (h_1(q|_{Q_1}, x|_{X_1}), h_2(q|_{Q_2}, x|_{X_2}))$$

- $Inv : \mathbf{Q} \rightarrow 2^{\mathbf{X} \times \mathbf{V}}$ given by:

$$Inv(q) = \left\{ (q, x) \in \mathbf{Q} \times \mathbf{X} : (q|_{Q_i}, x|_{X_i}, v|_{V_{ii}}, h_{ji}(q|_{Q_j}, x|_{X_j})) \in Inv_i, \text{ for } i, j = 1, 2, i \neq j \right\}$$

- $E \subset \mathbf{Q} \times \mathbf{Q}$ given by:

$$E = \left\{ (q, q') \in \mathbf{Q} \times \mathbf{Q} : ((q|_{Q_1}, q'|_{Q_1}) \in E_1 \wedge q|_{Q_2} = q'|_{Q_2}) \vee (q|_{Q_1} = q'|_{Q_1} \wedge (q|_{Q_2}, q'|_{Q_2}) \in E_2) \right\}$$

- $G : E \rightarrow 2^{\mathbf{X} \times \mathbf{V}}$ given by:

$$G(q, q') = \left\{ (x, v) \in \mathbf{X} \times \mathbf{V} : e_i = (q|_{Q_i}, q'|_{Q_i}) \in E_i \Rightarrow (x|_{X_i}, v|_{V_{ii}}, h_{ji}(q|_{Q_j}, x|_{X_j})) \in G_i(e_i) \right. \\ \left. \text{for } i, j = 1, 2, i \neq j \right\}$$

- $R : E \times \mathbf{X} \times \mathbf{V} \rightarrow 2^{\mathbf{X}}$ defined by:

$$R(q, q', x, v) = \left\{ x' \in \mathbf{X} : e_i = (q|_{Q_i}, q'|_{Q_i}) \in E_i \Rightarrow x'|_{X_i} \in R_i(e_i, x|_{X_i}, v|_{V_{ii}}, h_{ji}(q|_{Q_j}, x|_{X_j})) \right. \\ \left. \text{for } i, j = 1, 2, i \neq j \right\}$$

□

The sets of discrete and continuous, Q and X , variables of the composition automaton are the unions of the corresponding sets of the constituent automata. The set of input variables V is the union of the inputs of the constituent automata minus the union of their outputs. This is because there can be interaction between the constituent automata; an input of one of the automaton can be connected to an output of the other. The flow function f is defined by taking the interaction, if any, between the two constituent automata into account. When an input of one automaton is connected to the

output of the other it is not a free input any more but restricted by the output function h of the outputting automaton. Since the outputs of the constituent automata are disjoint, the output function h of the composition automaton is obtained simply by union of the output functions of the constituent automata. Invariants Inv are defined such that a transition is admitted in the composition if a transition is admitted in at least one of the constituent automata. Edges E are defined such that transitions are interleaving. Guards G are defined such that, a transition of the composition automaton is enabled if a transition is enabled in at least one of constituent automata. The reset function R is defined such that for a transition in the composition, a reset occurs in a constituent automaton if it is involved in the transition.

Using open hybrid automata, we can express assertions for components. The set of executions $Exec(OHA)$ of an open hybrid automaton $OHA = (Q, X, V, Y, Init, f, h, Inv, E, G, R)$ defines an assertion over an alphabet $P_{OHA} = Q \cup X \cup V \cup Y$. Also, using the alphabet equalization operator we introduced, we can cast the executions of OHA to any other alphabet. We define the assertion defined by an open hybrid automaton OHA over an alphabet P as $Asrt_P(OHA) = Pr_P^{\sim}(Exec(OHA))$. As we do for regular hybrid automaton, we require that an open hybrid automaton do not have any blocking executions for it to define a valid assertion. We will address non-blocking hybrid automata in Subsection 5.3.1.

5.3 Hybrid Observers

In Subsection 4.3.1, we presented a specification workflow which describes our system-specification process. In the specification workflow, initial requirements for a system are provided as a set of requirement items. Each requirement item describes a desired property of the system to be developed. The expected overall behaviour of the system is described collectively by all of the requirements. As stated in Subsection 4.3.1, these requirements are then formalized into contracts. First, each requirement is translated into a formal assertion. Then, contracts are formed by combining individual assertions. After the first contracts are formed, for the rest of the workflow, these contracts are refined, combined and assessed for validity. Each of these operations on contracts requires the individual assertions that make up the contracts to be combined and manipulated with set operations and checked against abstract system models.

In the previous section we described how one can express assertions using hybrid automata. Let us use an example in order to investigate the feasibility of the operations on contracts required by our workflow if we use hybrid automata to express assertions. Let SYS , H_1 and H_2 be three hybrid automata representing an abstract system model (SYS) for a component with alphabet P and two desired properties defined on P . Recall that, the assertions represented by SYS , H_1 and H_2 are denoted by $Asrt_P(SYS)$, $Asrt_P(H_1)$ and $Asrt_P(H_2)$. Now assume that we need to obtain union, intersection and complement of $Asrt_P(H_1)$ and $Asrt_P(H_2)$, and check if SYS satisfies the properties defined by those. These can be formulated as:

1. $Asrt_P(SYS) \subseteq Asrt_P(H_1) \cup Asrt_P(H_2)$

2. $Asrt_P(SYS) \subseteq Asrt_P(H_1) \cap Asrt_P(H_2)$
3. $Asrt_P(SYS) \subseteq Asrt_P(H_1) / Asrt_P(H_2)$

The first item can be formulated as the disjunction:

$$Asrt_P(SYS) \subseteq Asrt_P(H_1) \vee Asrt_P(SYS) \subseteq Asrt_P(H_2)$$

The second item can be formulated as the conjunction:

$$Asrt_P(SYS) \subseteq Asrt_P(H_1) \wedge Asrt_P(SYS) \subseteq Asrt_P(H_2)$$

Finally, the third item can be formulated as the conjunction:

$$Asrt_P(SYS) \subseteq Asrt_P(H_1) \wedge Asrt_P(SYS) \cap Asrt_P(H_2) = \emptyset$$

which can be represented using hybrid automata composition we defined in Definition 5.2.8 as follows:

$$Asrt_P(SYS) \subseteq Asrt_P(H_1) \wedge Asrt_P(SYS \parallel H_2) = \emptyset$$

In short, in addition to composition operation, we need a way to check refinement between two hybrid automata in order to effectively combine assertions represented by hybrid automata. In literature this problem is called as *trace inclusion* or *language inclusion* problem. However, using formal verification, it is not feasible to check inclusion between two hybrid automata. Even for timed automata, a restricted class of hybrid automata in which the continuous variables are limited to variables that keep the passage of time, inclusion or refinement relation between two automata has been shown to be undecidable for the general case (Alur and Madhusudan, 2004).

Actually, for all but a very restricted classes of hybrid automata, simulation is still the only tool feasible to use. In order to employ simulation for checking the refinement in our example, we can express both the system model SYS and the assertions H_1 and H_2 in a hybrid system simulation tool. Simulation, in this case, can be used for partial verification of the intended check. However, for instance, in order to check that SYS conforms to H_1 , one needs to explore the state spaces of both SYS and H_1 by simulation to see if all the executions of SYS are also the executions of H_1 :

$$\forall h \in Asrt_P(SYS) : h \in Asrt_P(H_1)$$

With simulation, checking if a particular hybrid trajectory is an execution of a hybrid automaton is difficult considering that the hybrid automaton can be non-deterministic.

For these reasons we choose to use *observers*¹, also called *monitors* in literature, to express assertions. A hybrid observer is an open hybrid automaton with its state space divided into ‘good’ and ‘bad’ regions. Hybrid observers are used to formalize *safety properties* of a system (Raskin, 2005). A hybrid observer’s function is to observe the shared variables of interest, which are provided as inputs, of a system model and partition its state space as ‘good’ and ‘bad’ regions. The reachability of the ‘bad’ states in

the composition of an observer with the model of the system under inspection enables to check if the system satisfies the assertion represented by the observer.

In order to explain use of observers we can use the earlier example. Let HO_1 and HO_2 be the reformulations of the assertions defined by H_1 and H_2 as hybrid observers. And let SYS be an open hybrid automaton modelling the system where all the ports of the system are provided as output variables of the automaton. Then we can check if SYS conforms to the assertions expressed by HO_1 and HO_2 by checking if there are any ‘bad’ states included in $Reach(SYS || HO_1)$ and in $Reach(SYS || HO_2)$.

Partial verification of these checks can be done using hybrid system simulation tools, as explained in Subsection 2.2.4, if the observers are *online testable*. With online testability we mean that an observer conforms to the definition of tests stated in Definition 2.2.1. Let HO_1 and HO_2 be the reformulations of the assertions defined by H_1 and H_2 as online-testable hybrid observers. Then, in a hybrid system simulation tool, we can explore the state space of composition of SYS with HO_1 and HO_2 to see if it is possible to reach a ‘bad’ state. If there happens to be an execution during the simulation that violates one of the assertions, the associated observer would go to a ‘bad’ state which would be indicated in its output. Otherwise, if all the simulations end without a ‘bad’ state alert, and if we are confident that we covered the state space of SYS sufficiently, the conformance of SYS to HO_1 and HO_2 is verified.

Furthermore, the set operations on assertions expressed by testable hybrid observers are also easy to define. Recall that every hybrid observer has an output that indicates whether the observer is in a ‘bad’ state or not. Then, since that the output value of a testable hybrid observer is deterministic, we can simply apply the logical operations *or*, *and* and *not* on the outputs of observers to combine them to construct new assertions. The *or*, *and* and *not* operations on observers correspond to the set operations *union*, *intersection* and *complement* applied on the sets of trajectories, i.e. assertions, they define.

5.3.1 Testable Hybrid Observers

In Subsection 5.2.1, we defined syntax and semantics of open hybrid automata. In this subsection, first, we define a hybrid observer as an open hybrid automaton. Then, we provide sufficient conditions for a hybrid observer to be online testable.

A hybrid observer can be defined as an extension of open hybrid automata definition given in Definition 5.2.4. All what needs to be done is to tag some locations as ‘bad’ locations.

Definition 5.3.1. A hybrid observer $HO = (Q, B, X, V, Y, Init, f, h, Inv, E, G, R)$ is an open hybrid automaton of which some locations are tagged as ‘bad’ locations defined by the set $B \subseteq Q$. □

¹ In control theory, a ‘state observer’ denotes a system providing an estimate of the internal state of a given real system, using the measurements of the input and output of the real system. Our use of term ‘observer’ is different and analogous to the notion of tests. In our context an observer starts with its output value as true and observes a run of a given system model. If the inspected run violates the property encoded in the observer, the observer turns its output value into false and stays that way.

Figure 5.2: Depiction of using a hybrid observer to test a system.

Using Definition 5.3.1, we can formally express properties which we would like to check on a hybrid automaton. Let OHA be an open hybrid automaton modelling a component to be developed and HO be a hybrid observer formalizing a property we would like OHA to satisfy. Then, assuming that OHA and HO are compatible, we can define the composition $OHA \parallel HO$, which has the set of executions $Exec(OHA \parallel HO)$. We can check if OHA satisfies the property formulated by HO , denoted by $OHA \models HO$, by checking the hybrid trajectories contained in $Exec(OHA \parallel HO)$. If $Exec(OHA \parallel HO)$ contains any trajectory along which a ‘bad’ location is visited, then we can conclude that H violates the given property, i.e. $OHA \not\models HO$. Otherwise OHA satisfies the given property, i.e. $OHA \models HO$.

To define an assertion for a component M with alphabet P , we can use a hybrid observer $HO = (Q, B, X, V, Y, Init, f, h, Inv, E, G, R)$ that has P as its input ports and has a single output port bad which indicates if the location is included in the set of ‘bad’ states defined by B :

- $V = P$
- $Y = \{bad\}$
- $h : bad = q \in B$

The assertion defined by such a hybrid observer are all the acceptable input traces of the observer, i.e. the traces that do not involve ‘bad’ states.

Definition 5.3.2. Given a hybrid observer HO , the assertion $Asrt(HO)$ of the hybrid observer is the set of all infinite input traces of HO along which a ‘bad’ location is not visited:

$$Asrt(HO) = pr_V \left(\left\{ (\tau, q, x, v, y) \in Exec(HO) \mid RANGE(q) \cap B = \emptyset \right\} \right)$$

where $RANGE(f)$ denotes the range of the function f . Note that, as opposed to denoting an assertion $Asrt_P(H)$ for a hybrid automaton H , while denoting the assertion $Asrt(HO)$ of a hybrid observer HO we omit the indication of the alphabet P of the specified component. This is because, in the case of a hybrid observer HO , the alphabet is already indicated by the set of input variables of HO : $V_{HO} = P$. $\square$

As we stated in the beginning of the section, in order to use a hybrid observer in a hybrid system simulation tool, we need the hybrid observer to be online testable, i.e. we need the hybrid observer to possess the properties of tests defined in Definition 2.2.1.

Focus on the hybrid observer HO depicted in Figure 5.2. Assume that we would like to run a set of simulations on SYS and HO and see if, during the simulations, SYS conforms to the assertion expressed by HO . For this, first, we need to be able to simulate HO for any set of trajectories observed on the input variables of HO , as long as we

like. Second, the value of the output of HO should be deterministic with respect to the inputs of HO .

In (Lygeros et al., 2008), the conditions for existence and uniqueness of solutions of hybrid automata are discussed. By ensuring the existence and uniqueness of its solutions of a hybrid observer, which is an open hybrid automata, we can ensure that it satisfies the above requirements. The two related properties for a hybrid automaton are being *nonblocking* and being *deterministic*. An open hybrid automaton is called nonblocking if there exists at least one infinite execution for every initial condition and for every input trajectory. An open hybrid automaton is called deterministic if there exists at most one maximal execution for every initial condition and for every input trajectory. These are the necessary and sufficient conditions for existence and uniqueness of solutions which allows for simulation of a property and instant diagnosis of a fault, i.e. a transition to a bad state. These requirements are summarized in Definition 5.3.3.

Definition 5.3.3. A testable hybrid observer $THO = (Q, B, X, V, Y, Init, f, h, Inv, E, G, R)$, is a hybrid observer which is nonblocking and deterministic. $\square$

In order to be able to construct a testable hybrid observer THO defined in Definition 5.3.3, we need to lay out sufficient conditions for THO to be nonblocking and deterministic. In order to do this, first, we present the sufficient conditions for regular hybrid automata defined in Definition 5.2.1 to be nonblocking and deterministic. Then, we modify the conditions for regular hybrid automata to achieve the sufficient conditions for open hybrid automata to be nonblocking and deterministic.

We call a regular (closed) hybrid automaton nonblocking if there exists at least one infinite execution for every initial condition. For a regular hybrid automaton H we define $Out(q)$ the set of points where continuous evolution is certain to violate the invariant $Inv(q)$ (Lygeros et al., 1999):

$$Out(q) = \{x' \in \mathbf{X} \mid \forall \delta > 0, \exists t \in [0, \delta) \text{ such that } x(t) \notin Inv(q) \text{ where } x(t) \text{ is the solution of } \dot{x} = f(q, x) \text{ with } x(0) = x'\} \quad (5.1)$$

Using $Out(q)$, we can define sufficient conditions for a hybrid automaton to be *nonblocking* as follows (Lygeros et al., 2008):

Definition 5.3.4. A hybrid automaton $H = (Q, X, Init, f, Inv, E, G, R)$ is *nonblocking* if:

1. $f(q, x)$ is *Lipschitz continuous*² in x for each $q \in \mathbf{Q}$.
2. For all $(q, x) \in Reach(H)$ with $x \in Out(q)$, $\exists (q, q') \in E$ such that $x \in G(q, q')$ and $R((q, q'), x) \neq \emptyset$.

$\square$

² Lipschitz continuity is a strong form of continuity for functions. A Lipschitz continuous function is limited in how fast it can change. For instance, a function $f : \mathbb{R} \rightarrow \mathbb{R}$ is called Lipschitz continuous if and only if there is a constant C such that $|f(x_1) - f(x_2)| \leq C|x_1 - x_2|$ for all x_1 and x_2 .

In Definition 5.3.4, the first item ensures that for each location there is a unique solution to the defined flow function. The second item states that there should be a discrete transition possible from the states where the state invariant is violated due to continuous evolution.

We call a regular hybrid automaton deterministic if there exists at most one maximal execution for every initial condition. The sufficient and necessary conditions for a hybrid automaton to be *deterministic* are as follows (Lygeros et al., 2008):

Definition 5.3.5. A hybrid automaton $H = (Q, X, Init, f, Inv, E, G, R)$ is *deterministic* if and only if for all $q, q', q'' \in Q$, and all $(q, x) \in Reach(H)$:

1. If $x \in G(q, q')$ for some $(q, q') \in E$, then $x \in Out(q)$.
2. If $(q', q'), (q'', q'') \in E$ and $(q' \neq q'')$, then $G(q, q') \cap G(q, q'') = \emptyset$
3. If $(q, q') \in E$ and $x \in G(q, q')$, then $R((q, q'), x)$ is a singleton.

□

In Definition 5.3.5, the first item states that there should be a discrete transition possible only when the continuous evolution is not possible. The second item states that, at any state, there can not be multiple discrete transitions enabled, with different target locations. The third item states that there should be only one reset position for a discrete transition.

We presented the sufficient conditions for regular hybrid automata to be nonblocking and deterministic. However, as our aim is to define testable hybrid observers, which are based on open hybrid automata (Definition 5.3.3), we need to modify these conditions for open hybrid automata. To see that we cannot use the same conditions in Definition 5.3.4 and Definition 5.3.5 for open hybrid automata, note that we cannot define $Out(q)$ defined in Equation (5.1) for the locations of an open hybrid automaton. The reason is, unlike a regular hybrid automaton, an open hybrid automaton possesses inputs variables controlled by the environment, values of which are unknown and can change instantly. Therefore, we cannot make a general definition for a set of points where, for a location q , $Inv(q)$ is certain to be violated with continuous evolution.

In order to tackle this problem, we propose to impose more determinism by using a variant of open hybrid automata which has *urgent transitions*. An urgent transition is a transition such that when enabled by the transition guard it should be taken immediately, without any delays. Also, urgent transitions have priority over the nonurgent transitions such that if in a state a number of urgent transitions are enabled as well as some nonurgent ones, one of the urgent transitions has to be taken.

We present a modification of open hybrid automata given in Definition 5.2.4 to define open hybrid automata with urgent transitions.

Definition 5.3.6. An open hybrid automaton with urgent transitions is a tuple $OHA_u =$

$(Q, X, V, Y, Init, f, h, Inv, E, E_u, G, R)$, where except for E_u are defined same as in Definition 5.2.4 and E_u is defined as follows:

- $E_u \subseteq E$ defines a set of urgent transitions as a subset of all transitions.
- Guards for all the urgent edges in E_u are closed sets.

□

Then, an execution of a open hybrid automaton with urgent transitions is defined as follows:

Definition 5.3.7. An *execution of an open hybrid automaton with urgent transitions* OHA_u is a hybrid trajectory $\chi = (\tau, q, x, v, y)$ satisfying:

1. Initial condition: $x_0(\tau_0) \in Init(q(\tau_0))$.
2. Discrete evolution: for all i , one of the following three conditions hold:
 - a) $(q_i(\tau'_i), x_i(\tau'_i)) = (q_{i+1}(\tau_{i+1}), x_{i+1}(\tau_{i+1}))$ and $(x_i(\tau'_i), v_i(\tau'_i)) \notin \bigcup_{e \in E_u(q_i(\tau'_i))} G(e)$
 - b) $e_i = (q_i(\tau'_i), q_{i+1}(\tau_{i+1})) \in E_u$, $(x_i(\tau'_i), v_i(\tau'_i)) \in G(e_i)$, and $x_{i+1}(\tau_{i+1}) \in R(e_i, x_i(\tau'_i), v_i(\tau'_i))$.
 - c) $e_i = (q_i(\tau'_i), q_{i+1}(\tau_{i+1})) \in E/E_u$, $(x_i(\tau'_i), v_i(\tau'_i)) \notin \bigcup_{e \in E_u(q_i(\tau'_i))} G(e)$, $(x_i(\tau'_i), v_i(\tau'_i)) \in G(e_i)$ and $x_{i+1}(\tau_{i+1}) \in R(e_i, x_i(\tau'_i), v_i(\tau'_i))$.
3. Continuous evolution: for all i with $\tau_i < \tau'_i$:
 - a) $q_i : I_i \rightarrow Q$ is constant over $t \in I_i$, i.e. $q_i(t) = q_i(\tau_i)$ for all $t \in I_i$;
 - b) $x_i : I_i \rightarrow X$ is the solution to the differential equation

$$\frac{dx_i}{dt} = f(q_i(t), x_i(t), v_i(t))$$

over I_i starting at $x_i(\tau_i)$; and,

- c) for all $t \in [\tau_i, \tau'_i]$, $(x_i(t), v_i(t)) \in Inv(q_i(t))$ and $(x(\tau'_i), v(\tau'_i)) \notin \bigcup_{e \in E_u(q(\tau'_i))} G(e)$

4. Output evolution: for all i and for all $t \in \tau_i, y_i(t) = h_i(q_i(t), x_i(t))$

□

Note that, according to the second item in Definition 5.3.7 on discrete transitions, there are three types of transitions possible:

- As defined in 2.a, an environmental transition can occur, taking place outside of the open hybrid automaton and affecting only the input variables, as long as an urgent transition is not enabled. An urgent transition has priority over an environmental transition.
- As defined in 2.c, an urgent internal transition can occur.
- As defined in 2.b, a non-urgent internal transition can occur, where the state changes, as long as an urgent transition is not enabled. An urgent transition has priority over a non-urgent transition.

Also, the item 3.c states that a continuous evolution is only possible if an urgent transition is not allowed; an urgent transition cannot be delayed. Note also in Defini-

tion 5.3.6 we require the guards for the urgent edges to be closed sets. The hybrid trajectories for a hybrid automata with urgent edges that have open guard sets are undefined. This is due to that at the boundary of an interval of a guard, the edge is not enabled and when we pass that boundary we are delaying. The point of transition is undefined.

Compositions of two OHA_u 's can be defined by modifying Definition 5.2.8.

Definition 5.3.8. The *composition* of two open hybrid automata with urgent transitions OHA_{u_1} and OHA_{u_2} is a hybrid automaton $OHA_{u_{1,2}} = OHA_{u_1} \parallel OHA_{u_2} = (Q, X, V, Y, \Sigma, Init, f, h, Inv, E, E_u, G, R)$, with all the definitions are the same as in Definition 5.2.8 except for E . For $OHA_{u_{1,2}}$, E and E_u are defined as follows:

- $E_{u_{1,2}} \subseteq \mathbf{Q} \times \mathbf{Q}$ is given by

$$E_{u_{1,2}} = \{ (q, q') \in \mathbf{Q} \times \mathbf{Q} : ((q|_{Q_1}, q'|_{Q_1}) \in E_{u_1} \wedge q|_{Q_2} = q'|_{Q_2}) \vee (q|_{Q_1} = q'|_{Q_1} \wedge (q|_{Q_2}, q'|_{Q_2}) \in E_{u_2}) \}$$

- $E = (E \setminus E_{u_{1,2}}) \cup E_{u_{1,2}} \subseteq \mathbf{Q} \times \mathbf{Q}$ where $E \setminus E_{u_{1,2}}$ is given by

$$E \setminus E_{u_{1,2}} = \{ (q, q') \in \mathbf{Q} \times \mathbf{Q} : ((q|_{Q_1}, q'|_{Q_1}) \in E_1 \setminus E_{u_1} \wedge q|_{Q_2} = q'|_{Q_2}) \vee (q|_{Q_1} = q'|_{Q_1} \wedge (q|_{Q_2}, q'|_{Q_2}) \in E_2 \setminus E_{u_2}) \}$$

□

According to Definition 5.3.8, in the composition automaton of two open hybrid automata, an urgent transitions is enabled if there is an urgent transition is enabled in at least one of the automata. On the other hand, a non-urgent transition is enabled if no urgent transition is enabled in any of the automata and a non-urgent transition is enabled in at least one of the automata. Looking back to Definition 5.3.7, composition of two open hybrid automata OHA_{u_1} and OHA_{u_2} with urgent transitions where there is some interaction between the two, i.e. $((Y_1 \cap V_2) \cup (V_1 \cap Y_2)) \neq \emptyset$, is well defined only if either one of the automaton has an empty set of urgent transitions: $E_{u_1} = \emptyset$ or $E_{u_2} = \emptyset$. If OHA_{u_1} and OHA_{u_2} interacts with each other and they both have non-empty sets of urgent transitions, then according to 2.a of Definition 5.3.7 there will be a deadlock whenever both automata have urgent transitions enabled at the same time. However, this limitation is not harmful to our case since we only need to compose an observer, which has urgent transitions, with a system model which does not.

The following definition lists the sufficient conditions for an open hybrid automaton with urgent transitions to be nonblocking and deterministic.

Definition 5.3.9. An open hybrid automaton with urgent transitions OHA_u is non-blocking and deterministic if:

1. For every location $q \in \mathbf{Q}$, $f(q, x, v)$ is Lipschitz continuous in x and piecewise continuous in v .
2. For every location $q \in \mathbf{Q}$, $Inv(q) = \mathbf{X} \times \mathbf{V}$.
3. All transitions are urgent, $E_u = E$.

4. For every $q \in \mathbf{Q}$, the edges in $E(q, \cdot)$ have mutually exclusive guards, i.e. if $e_1 = (q, q_1), e_2 = (q, q_2)$ and $q_1 \neq q_2$, then $G(e_1) \cap G(e_2) = \emptyset$.
5. If $e = (q, q') \in E$ and $(x, v) \in G(e)$, then $R(e, x, v)$ is a singleton.

□

The items 1 and 2 in Definition 5.3.9 ensure that OHA_u is nonblocking. The item 1 ensures that within a location $q \in \mathbf{Q}$, $\dot{x} = f(q, x, v)$ always have a solution. The item 2 ensures that within a location $q \in \mathbf{Q}$, flow is unrestricted. The item 2 is required due to the input variables of open hybrid automata preventing to make a general definition for a set of points where invariant of a location q is certain to be violated with continuous evolution. Such a definition ($Out(q)$) was essential for ensuring non-blockingness of regular hybrid automata (Definition 5.3.4).

The items 3, 4, and 5 in Definition 5.3.9 ensure that OHA_u is deterministic. The item 3 ensures that when a transition is enabled it is taken without any continuous evolution. This condition is again essential for open hybrid automata since we lack $Out(q)$ function used in Definition 5.3.5 for regular hybrid automata.

The item 4 ensures that no two transitions are enabled at the same time. And, the item 5 ensures that a transition there is only one reset position for a transition.

An open hybrid automaton formulated according to Definition 5.3.9 can be simulated. However, it might have Zeno behaviour meaning that it can enter an endless loop of internal transitions, which results in an inconclusive test. For an OHA_u to be used as a test it needs to be *I/O feasible* as well. In (Lynch et al., 2003) I/O feasibility is defined as the property for hybrid automata with inputs and outputs that “implies that any finite execution fragment can be extended to an admissible execution in response to any admissible input from the environment” (See Definition 5.1.2 for the definition of admissibility.) This means that an I/O feasible open hybrid automaton cannot exhibit Zeno behaviour as long as the input trajectory does not. An I/O feasible open hybrid automata is nonblocking by definition.

(Lynch et al., 2003) presents a detailed discussion for the *receptiveness* property of Hybrid I/O Automata (HIOA) which ensures I/O feasibility. For our definition of OHA , which is defined in a more explicit way than HIOA, we have not formulated conditions that ensures I/O feasibility. (Lygeros et al., 2008) presents a discussion of types of Zeno behaviours and potential resolutions of Zeno phenomena. While constructing observers in our approach, one can prevent or resolve Zeno phenomena using the guidance provided in (Lygeros et al., 2008).

5.3.2 Assertions Expressed with Testable Hybrid Observers

In the beginning of Section 5.3, we already stated that with hybrid observers one can only express assertions in the form of safety properties. However, it is still an open question for us whether one can express any kind of safety property with a testable observer or a combination of testable observers formulated according to Definition 5.3.9. Note also that the assertions obtained by complementing and/or combining hybrid observers might not be safety properties. For the rest of the discussion we assume

that for a component, we know how to express any kind of safety property with a combination of testable observers.

As we stated in the beginning of the chapter we specify the behaviours of a component with an A/G contract. Let M be a component with an alphabet P with a variable $v \in P$. And let TO be a testable observer over P formulated according to Definition 5.3.9. Then, it is not apparent from TO whether v is an input or an output for M , i.e. whether v is in control of the environment of M or of M itself. This is because TO , and any observer for that matter, does not differentiate between the inputs and outputs of the component. All the variables in P are inputs of TO . Input/output status of the ports for M need to be stated explicitly along with P . This can be done by defining two disjoint subsets P_i and P_o of P such that $P = P_i \cup P_o$ which identify input and output ports M . However, some ports in P might not be suitable to be strictly classified as inputs or outputs. For instance, take the environment variables in V_e defined in our specification model in Subsection 4.1.1. Some of the variables in V_e would be under the influence of both the environment and the system. In this direction we only require that the union of the sets of input and output ports to be a subset of all the ports: $P \supseteq P_i \cup P_o$.

The classification of inputs and outputs of a component restricts the definition of an associated A/G contract. Let $C_M = (A_M, G_M)$ be a contract for the component M with alphabet $P \supseteq P_i \cup P_o$. According to Section 5.1, A_M and G_M are assertions in the form of sets of trajectories over P . Intuitively, the assumption assertion A_M cannot constrain the output ports in P_o and, similarly, the guarantee assertion G_M cannot constrain the input ports in P_i .

This requirement can be formalized as follows:

Definition 5.3.10. Let $C = (A, G)$ be an A/G contract defined for a component M with alphabet P and disjoint input and output sets P_i and P_o ($P_i \cup P_o \subseteq P$). The assumption assertion A and the guarantee assertion G cannot constrain the outputs and inputs of M , respectively:

$$pr_{P_o}(A) = Traj^\infty(P_o)$$

$$pr_{P_i}(G) = Traj^\infty(P_i)$$

□

Note that Definition 5.3.10 does not prevent specifying relations between inputs and outputs. For instance, a guarantee assertion can still define value of an output variable based on a input variable.

For each individual assertion that is expressed by a single observer, one of the two requirements in Definition 5.3.10 should hold depending on whether the assertion is an assumption or a guarantee. However, it is hard to check these requirements on an assertion if it is constructed as a combination of other assertions which are expressed by observers. Out of all the set operations, which are *union*, *intersection* and *complement*, that can be used to combine assertions, only the *union* operation preserves these two requirements. This is since *intersection* and *complement* operations might result in

less number of traces and the resulting assertion might violate Definition 5.3.10. So if A_M or G_M is defined as a combination of assertions during which intersection or complement operations are used, then it is possible that A_M or G_M do not conform the requirements. In order to partially verify that A_M and G_M conform to the requirements in Definition 5.3.10, one can use simulations on the combinations of observers that make up A_M and G_M .

5.4 Methods for Hybrid Observers

In Section 5.1, we defined the concrete formalism for assertions of A/G contracts used in system specification. In the previous section, we defined testable hybrid observers which are used to express the assertions in a testable format. In this section, we describe the practical ways of using hybrid observers as tests in hybrid system simulation tools.

We stated in Section 4.3 that, in the specification phase, validation of specification contracts is vital to achieve good quality specifications. While validating contracts, the fundamental operation is to check whether a particular behaviour conforms to an assertion. As we stated in Section 5.3, simulation is the only feasible method (as opposed to formal verification) we can employ on contracts that include hybrid behaviours. In this direction, we described how to express assertions on hybrid behaviours using hybrid observers which are online testable.

In this section, we describe how to implement and simulate a hybrid observer in a hybrid system simulation tool, 20-sim. Our choice of 20-sim as the demonstration tool is motivated by the fact that our models are implemented in 20-sim format. Although the specifics of implementation and simulation of observers depend on the syntax and semantics of a particular tool, the method and concerns we describe on 20-sim are mostly generalizable to any hybrid system tool. Our reason to choose a tool independent formalism for hybrid observers, i.e. hybrid automata, is motivated by the desire to leave it open to which tool to use and to avoid the not fully-defined and changeable semantics of the tools.

5.4.1 Implementing Observers in 20-sim

In 20-sim there is no way to directly implement observers or, more simply, any sort of tests. However, one can easily implement blocks in 20-sim with observer functions, i.e. monitoring a set of ports and outputting a boolean value. This way any hybrid observer can be translated into 20-sim format.

In order to demonstrate the process of implementing a hybrid observer in 20-sim, we give a simple hybrid observer example in Example 5.4.1.

Example 5.4.1. In Figure 5.3 a hybrid observer, which we will call HO , is shown. HO is the tuple $(Q_{HO}, B_{HO}, X_{HO}, V_{HO}, Y_{HO}, Init_{HO}, f_{HO}, h_{HO}, Inv_{HO}, E_{HO}, G_{HO}, R_{HO})$. HO is formed for a component with alphabet $P = V_{HO} = \{v, d\}$. The domains for the variables in P are defined as $D(v) = \mathbb{R}$ and $D(d) = \{on, off\}$. HO has three discrete locations, $|Q|_{HO} = 3$, and a single continuous variable, $X_{HO} = \{x\}$. The assertion that HO formulates requires that when $d = on$ the value of v should stay within a certain range

Figure 5.3: An example hybrid observer.

with respect to x : $v \geq x - 5 \wedge v \leq x + 5$. When d becomes *on*, a transition to the location *S1* occurs, x is reset to 0 and from that point x increases with a rate of 3. In Figure 5.3, if v leaves the proximity of x , i.e. $v < x - 5$ or $v > x + 5$, while $d = \text{on}$, then a transition to the ‘bad’ location occurs. When $d = \text{off}$ no constraint is imposed on v . $\square$

In Example 5.4.2, we present the 20sim implementation of the hybrid observer from Example 5.4.1.

Example 5.4.2. In Listing 5.1, the 20sim implementation of the hybrid observer HO introduced in Example 5.4.1 is shown. HO is defined as an equation submodel in 20sim. In 20sim, definitions in an equation submodel are collected under distinct sections specific to the type of the definitions. HO definition is five sections: *submodel*, *constants*, *variables*, *initialequations* and *code*. Within the *submodel* section (lines 1-7), the ports corresponding to the input (V_{HO}) and output (Y_{HO}) ports of HO are declared. There are two input ports, v and d , as in V_{HO} . There is one output port, *output*, which outputs 0 if HO has gone into the bad location, and 1 otherwise. In the *variables* section (lines 16-24), variables are declared. The discrete location of HO is represented by the variable *location*. The variable x represents the internal continuous variable x in X_{HO} . In the *constants* section (lines 9-14), the constants are declared. The three locations in Q_{HO} are represented by the integer constants *INITIAL_LOCATION*, *S1_LOCATION* and *BAD_LOCATION*. The set of discrete values for d are represented by the integer constants *ON* and *OFF*. In the *initialequations* section (lines 26-28), the initializations *location* = *INITIAL_LOCATION* and $x = 0$ that correspond to $Init_{HO}$ are made.

In the *code* section (lines 30-68), the code body of HO that is to be executed at every simulation step is defined. In the event part (lines 31-34) of the *code* section, the events that trigger the discrete transitions are defined. These events correspond to the guard functions in G_{HO} . To define the events the *event* function of 20sim is used. The *event* function simply forces the integration algorithm of the simulation engine to find the exact point where the value of the given argument crosses zero. The assignment to the variable *d_change* is used to force the integration algorithm to catch a change in the discrete variable d is detected. The assignments to the variables *v_out1* and *v_out2* are used to catch the exact times when $v < x - 5$ or $v > x + 5$. The discrete transition part (lines 36-51) handles the discrete transitions in E_{HO} . Note that if the location is the bad location (*location*=*BAD_LOCATION*) then the observer stays there, there is no outgoing transition. In the continuous evolution part (lines 53-68), the code corresponding to the continuous evolution function f_{HO} and the output function h_{HO} are

<pre> 1 submodel HO 2 type Submodel 3 ports 4 signal in v; 5 signal out output; 6 signal integer in d; 7 end; 8 9 constants 10 integer INITIAL_LOCATION = 0; 11 integer BAD_LOCATION = -1; 12 integer S1_LOCATION = 1; 13 integer ON = 1; 14 integer OFF = 2; 15 16 variables 17 integer location; 18 real x; 19 real I_dx; 20 real B_dx; 21 real S1_dx; 22 boolean d_change; 23 boolean v_out1; 24 boolean v_out2; 25 26 initialequations 27 location = INITIAL_LOCATION; 28 x = 0; 29 30 code 31 // event part 32 d_change = event(d - ((ON+OFF)/2)); 33 v_out1 = event(v - (x-5)); 34 v_out2 = event(v - (x+5)); </pre>	<pre> 35 // discrete transition part 36 switch location 37 case INITIAL_LOCATION do 38 if d == ON then 39 location = S1_LOCATION; 40 x = 0; 41 S1_dx = 0; 42 end; 43 case S1_LOCATION do 44 if d == OFF then 45 location = INITIAL_LOCATION; 46 else 47 if v < x-5 or v > x+5 then 48 location = BAD_LOCATION; 49 end; 50 end; 51 end; 52 53 // continuous evolution part 54 I_dx = int(0,x); 55 S1_dx = int(3,x); 56 B_dx = int(0,x); 57 58 switch location 59 case INITIAL_LOCATION do 60 x = I_dx; 61 output = 1; 62 case S1_LOCATION do 63 x = S1_dx; 64 output = 1; 65 case BAD_LOCATION do 66 x = B_dx; 67 output = 0; 68 end; </pre>
--	---

Listing 5.1: The 20-sim implementation of the observer in Figure 5.3.

defined. In the lines 54-56, the continuous evolutions of x in each location of HO are defined using the integration function *Int* of 20sim. □

We can use the Listing 5.1 to define a template for implementing a hybrid observer in 20sim. In Table 5.1, a guide to translate a hybrid observer into a 20sim model is presented. On the left side, the sections and parts of the 20sim implementation are listed. On the right of each section or part, the definitions to be made in that section/part related to the parts of the formal definition of the hybrid observer are summarized.

<i>Sections and parts of the 20sim observer implementation</i>		$HO = (Q, B, X, V, Y, Init, f, h, Inv, E, G, R)$
ports section		All the ports in V and Y are declared.
constants section		The discrete locations in Q are declared as integer constants. Bad locations in B are stated.
variables section		An integer variable to store the discrete location is defined. For every variable in X , a corresponding real variable is defined.
initialequations section		The initial location and the initial values of the continuous variables in X , as defined by $Init$, are defined.
code section	event part	The events corresponding to the enabling of the transitions guards in G are defined.
	discrete transition part	The discrete transitions defined by the list of transitions E , the transition guards G and the reset function R are defined.
	continuous evolution part	The continuous evolution function f and the output function h are defined.

Table 5.1: Mapping of hybrid observer parts to the sections and parts of 20sim observer implementation.

Figure 5.4: A source model and hybrid observer model connected in 20sim.

5.4.2 Simulating Observers

In the previous subsection we provided a method to translate a hybrid observer into a 20sim model. In this subsection we describe how single observers or combinations of them can be used in simulations.

An observer is a form of test. It is made to observe some variables of a given system model for ‘bad’ states, declaratively stating properties (safety properties in this case) for the specified system. Therefore, we cannot execute an observer alone, i.e. directly run simulations on it. In order to assess validity of a single observer or a combination of observers in a hybrid system simulation tool, we need to feed them with behaviours. This is done by connecting an observer’s input ports to output ports of a model that acts as a signal source, which can be an abstract or partial model of the specified system. Then the composed model can be simulated.

Figure 5.5: A simulation of the observer from Example 5.4.2.

Figure 5.6: An observer defined as a logical combination of three other observers.

Example 5.4.3. In Figure 5.4, a 20sim model is shown where the inputs of the 20sim observer model from Example 5.4.2 is connected to the output of a source model. In Figure 5.5, the plot of a simulation of this combination is depicted. In the figure, plots of 4 variables are presented: the internal continuous variable x , the continuous input v , the discrete input d and the location variable $location$. The source component defines a behaviour for the input variables v and d of the observer. In this behaviour v is constant at 3 whereas d starts with *OFF* and then switches to *ON* after 4 seconds, when $t = 4$. At $0 \leq t \leq 4$, the observer stays at the initial location. At $t = 4$ it switches to the *S1* location where x is reset to 0 and starts to evolve according to $\dot{x} = 3$ until $x = 8$. Right after $x > 8$ the guard of the transition to the bad location is enabled since $x - 5 > v$, at which point the observer goes to the bad location. $\square$

In a hybrid system simulation tool, combinations of observers using Boolean operations, which correspond to combinations of the associated assertions using set operations, can be modelled by taking the boolean combinations of the outputs of the observers.

Example 5.4.4. Let HO_1 , HO_2 and HO_3 be three observers which are individually implemented in 20sim. Then the combination $HO_C = (HO_1 \vee HO_2) \wedge \neg HO_3$ of these observers, which corresponds to the combination $(Asrt(HO_1) \cup Asrt(HO_2)) \setminus Asrt(HO_3)$ of their respective assertions, can be modelled as shown in Figure 5.6. $\square$

As we want to test observers with simulations, we would like the test to come to halt as soon as the test result is certain to be negative or positive. For instance, while simulating HO_C , we want the simulation to stop with HO_C giving a negative output value as soon as both HO_1 and HO_2 get into bad locations. In all other cases, the simulation should continue running as long as user does not intervene. To this end, we use a four-valued logic, Boolean logic extended with auxiliary inconclusive truth values:

- T : True
- F : False
- IT : Indeterminate-true
- IF : Indeterminate-false

The auxiliary values IT and IF represent the indeterminate states of a boolean result as individual outputs of observers or the values derived from them can be indeterminate during a simulation. In contrast to three-valued logic used in model-based testing (Utting et al., 2012), which has a single indeterminate truth value to represent an inconclusive test verdict, we have two types of indeterminate truth values. The reason is that, for an ongoing or stopped simulation with an inconclusive test result, we want to have an indication, i.e. T in IT and F in IF , that states the result of the test if all the outputs of all the involved observers stay the same for an infinite amount of time.

The binary logical operations, i.e. *and*, *or* and *negation*, in our four-valued logic are intuitively defined as follows. If an operation involves only non-auxiliary truth values, i.e. T and F , then the result is the same with Boolean-logic. If an operation involves only auxiliary values, i.e. IT and IF , then the result is an indeterminate value with the truth indicating part derived by applying the operation to the truth indicating parts of the operands. An example is $IF \wedge IT = IF$. If an operation involves both a determinate value and an indeterminate value, then the result can be determinate or indeterminate. The result is determinate, if it can be derived by just looking at the determinate value, e.g. $T \vee IF = T$. Otherwise the result is indeterminate with the truth indicating part derived by applying the operation to the truth indicating parts of the operands, e.g. $F \vee IF = IF$.

In a simulation, an individual observer starts with its output value IT . If the observer goes to a bad location, its output becomes F . If output of an observer or of a combination of observers obtains a determinate value during the simulation then it stays that way. If the test ends while output of an observer or of a combination of observers has an indeterminate value, it stays that way.

Example 5.4.5. For HO_C of Example 5.4.4, the four-valued truth table is shown in Table 5.2. At the beginning of any simulation the outputs of all observers are IT corresponding to rows 2 and 3. For HO_C to obtain a determinate value, both HO_1 and HO_2 go into bad locations as in row 1. If, during a simulation, HO_3 goes into a bad location and either HO_1 or HO_2 does not, then the value of HO_C is IT as in rows 4 and 5. $\square$

Simulation of assertions formulated as hybrid observers can only assist in partial verification of assertions on a system model. Due to the continuous variables in the input traces, the state space can not be exhaustively covered with simulation. The idea with partial verification is that, user can gain confidence by running various simulations with different input traces. Furthermore, due to infinite traces, as we have seen in Example 5.4.5, for a given assertion modelled by hybrid observers, there will be combinations of outputs of observers that do not yield a determinate result during simulation. At this point, having indeterminate results with truth value indications provides the user with an insight of the result when the user believes a particular set of input traces are covered in a simulation but the result is still indeterminate. User might run the simulation until the point where the result of all observers are decided, by the user, to be the same if the user continued the simulation according to the intended input trace.

	HO_1	HO_2	HO_3	HO_C
1	F	F	$\times$	F
2	$\times$	IT	IT	IF
3	IT	$\times$	IT	IF
4	$\times$	IT	F	IT
5	IT	$\times$	F	IT

Table 5.2: Four-valued truth table for $HO_C = (HO_1 \vee HO_2) \wedge \neg HO_3$.

5.4.3 Syntax and Semantics of Hybrid System Simulation Tools

Each hybrid system simulation tool has its own syntax and semantics for its input language. Syntax and semantics of a hybrid system tool can be different than of the hybrid observer formalism we introduced in Section 5.3. See (Carloni et al., 2006) for detailed discussion of syntax and semantics of popular hybrid system simulation tools. Furthermore, in a hybrid system tool, there can be multiple implementations of the semantics by the simulation engine. For instance, the 20sim tool has 10 different integration algorithms that can be used for a simulation. Selection of the integration algorithm can affect the simulation results and some algorithms might be more suited for simulation of a model than others. For these reasons, one should be aware of the syntax and semantics of the simulation tool used in order to construct/translate an observer model correctly, to pick correct simulation parameters and to draw correct results from the simulations.

5.5 Validity of Assume/Guarantee Contracts

In Subsection 4.3.2 we defined three validity criteria for specification contracts: correctness, completeness and consistency. We explained the meaning of these criteria for each of the items, i.e. the A/G contracts and requirements, involved in the workflow described in Subsection 4.3.1. In this section we explain what each validity criterion means for the concrete formalism of A/G contracts we defined in this chapter, i.e. A/G contracts with hybrid behaviours. We also explain the methods that can be utilized to assess validity of the contracts.

In the specification workflow (Subsection 4.3.1), a contract C can be classified into one of the three following classes according to the way it is derived or constructed:

1. C is constructed from a set of requirements R . Items in this class are the first versions of the mode contracts $C_{m_1}, \dots, C_{m_n}$ and the switching contract C_{sw} which are directly constructed using the mode requirements $R_{m_1}, \dots, R_{m_n}$ and the switching requirements R_{sw} .
2. C is derived from another contract C' via modification. Items of this class are all the contracts which are derived by modifying the individual mode contracts $C_{m_1}, \dots, C_{m_n}$, the mode-composition contract C_{mc} , the mode-switching contract C_{sw} or the overall system contract C_{sys} .
3. C is constructed by direct composition of a number of contracts $C_1, \dots, C_n$. Items

in this class are the first versions of the mode-composition contract C_{mc} and the overall system contract C_{sys} .

5.5.1 Correctness

As stated in Subsection 4.3.2, for a contract C , we define correctness as the refinement relation between C and the contract(s) or the set of requirements that precedes C in the specification workflow shown in Figure 4.8.

If C belongs to the first class defined above, C is preceded in the workflow by a set of requirements R , the type of reference for correctness is the requirements. A set of requirements, as opposed to an A/G contract, does not have a sharp environment/-component distinction. It just defines a set of behaviours on a set of variables without identifying the responsibilities of the component or the environment. For a set of requirements R , we denote the set of behaviours defined by R by B_R . Here we use the same notation for B_R as with the obligations of a contract we defined in Subsection 2.2.3 and call it the obligations of R since the meaning is the same. For $C = (A, G)$ to refine the set of requirements R means that the obligations of C is a subset of obligations of R :

$$C \leq R : B_C = A \cap G \subseteq B_R$$

In order to verify this relation, one can run simulations on the observer models of A and G in a hybrid system simulation tool. For the relation to hold, every behaviour that is accepted by $A \cap G$ should be also accepted by R . If we have the observer model for $A \cap G$ is 20-sim then we need to show:

$$\forall \chi \in \text{Asrt}(HO_{A \cap G}) : \chi \in B_R$$

Note that usually R is an informal description such that it is not executable. Therefore checking if a particular behaviour is within B_R has to be a manual and informal task.

If C belongs to second class, i.e. C is preceded by another contract in the workflow, then the reference for correctness is another formal A/G contract $C' = (A', G')$. Through the specification workflow, the set of behaviours (termed as obligations) defined by the requirements in the beginning have to be preserved. For this reason for C to be correct with respect to C' , it has to preserve its obligations, which is called ‘conformance’ (Subsection 4.3.2):

$$B_C \subseteq B_{C'} : (A \cap G) \subseteq (A' \cap G')$$

For C to be correct it also has to refine C' . It might relax the assumptions and strengthen the guarantees of C' :

$$C \leq C' : A \supseteq A' \wedge G \subseteq G'$$

Note that the refinement relation preserves the separation of environment/component responsibilities made by C' . That a contract C both conforms and refines another

contract C' is called 'strong refinement' and denoted by $C \leq_s C'$ (Subsection 4.3.2). So we can say the C is correct with respect to C' if it strongly refines C' :

$$C \leq_s C' : C \leq C' \wedge B_C \subseteq B_{C'}$$

The above relation can be partially verified with simulations on the observer models of A , G , A' and G' by verifying:

1. $\forall \chi \in \text{Asrt}(HO_{A'}) : \chi \in \text{Asrt}(HO_A)$
2. $\forall \chi \in \text{Asrt}(HO_G) : \chi \in \text{Asrt}(HO_{G'})$
3. $\forall \chi \in \text{Asrt}(HO_{A \cap G}) : \chi \in \text{Asrt}(HO_{A' \cap G'})$

If C belongs to the third class, i.e. if C is obtained directly from composing other contracts, then the correctness of C does not have to be assessed since C is equal to the composition and correct by definition.

5.5.2 Completeness

As described in Subsection 4.3.2, the reference for completeness $\kappa(C)$ of a specification contract C is the external entity that C specifies. Since such an external entity is always incompletely and informally defined, for which the specification process aims to obtain a formal or semi-formal and complete specification, the completeness $\kappa(C)$ of the contract C is always an informal property. Therefore, we say that the aim of completeness related activities is to achieve sufficient completeness.

Assume that C belongs to first class, i.e. C is constructed from a set of requirements R , and that C is correct with respect to R . As stated in Subsection 4.3.2, we do not assess completeness of requirements, so we do not have any information on completeness of R at the point when C is being constructed. The completeness of C has to be assessed using simulations on the observer models HO_A and HO_G of C . There is not a formula for this assessment since the reference for completeness would be informal and incomplete.

Regarding completeness of R , we can deduce it from completeness of C . The completeness of C implies the completeness of R since C is correct with respect to R :

$$C \leq R \Rightarrow (\kappa(C) \Rightarrow \kappa(R))$$

Now assume that C belongs to the second class, i.e. C is preceded by another contract C' in the workflow, and that C is correct with respect to C' . Also assume that C' is shown to be complete, so $\kappa(C')$ holds. Despite that C' is complete, it is still possible that C is not complete. This is because, according to strong refinement relation $C \leq_s C'$ required for correctness of C , C might omit some behaviours contained in C' , i.e. $B_C \subseteq B_{C'}$, and this might cause incompleteness of C . For this reason, completeness of C has to be assessed on the observer models HO_A and HO_G for C via simulations.

Now assume that C belongs to the third class, i.e. C is obtained directly from composing other contracts: $C = C_1 \otimes \dots \otimes C_n$. Assume also that C is consistent and all the constituent contracts are complete: $\kappa(C_1) \wedge \dots \wedge \kappa(C_n)$. Still it is possible for C to be incomplete. This is because an incompleteness in C might not be possible to identify

in its constituent contracts; sufficient completeness of the constituent contracts does not imply sufficient completeness of the composition. For this reason completeness of C should be assessed using simulations on the observer models for C which are combinations of the observer models for $C_1, \dots, C_n$.

5.5.3 Consistency

According to Subsection 2.2.2 and our definition of consistency in Subsection 4.3.2, which deals with consistency of a contract regarding both the environment and the component, given that $C = (A, G)$ is a saturated contract, then C is consistent if it defines feasible legal environments, i.e. $A \neq \emptyset$, and feasible implementations, i.e. $G \neq \emptyset$. However, in such a case, environments and implementation of a contract C might still be inconsistent with each other, i.e. $B_C = A \cap G = \emptyset$. Therefore, for consistency, we require C to satisfy the stronger condition of having non-empty behaviours. A saturated contract C is called consistent if and only if it defines consistent implementations and legal environments, i.e. $B_C \neq \emptyset$.

For a contract C that belongs to any of the three classes defined in the beginning of this section, we need to assess consistency using the simulations on the observer models for C . In order to verify the consistency condition the following should be shown with simulations:

$$\exists \chi : \chi \in \text{Asrt}(HO_{A \cap G})$$

5.6 Conclusions

In this chapter we presented a concrete formalism for A/G contracts to be used in system specifications. First, we gave a formal definition of a hybrid behaviour over a set of variables. Then, we defined assumption and guarantee assertions of contracts to be sets of hybrid behaviours which specify the behaviours of the specified component and of the environments suitable to host the specified component. Having a formal definition of assertions, i.e. having formal contracts, is vital to avoid ambiguities in specifications.

Aimed at using hybrid system simulation tools to assess the validity of specifications, we proposed employing testable hybrid observers to express assertions. A hybrid observer is an open hybrid automaton that defines properties on the behaviours over a set of ports. A testable hybrid observer is defined as a deterministic and nonblocking hybrid observer. Our contribution at this point is, inspired by the conditions for a regular hybrid automaton to be deterministic and nonblocking stated in literature, we defined the formalism of open hybrid automata with urgent transitions to arrive at testable hybrid observers, which are based open hybrid automata.

That our hybrid observers are testable enables them to be used as tests in simulations. The automata-based syntax of assertions might make it harder to define and understand them, however, this issue can be mitigated by using a different syntax with the same underlying semantics.

We favour expressing assumption and guarantee assertions of a contract with combinations of smaller and independent observer-based assertions. This approach en-

ables a direct mapping of assertions to the requirements and, hence, enhances requirements traceability. Also, formulating assertions, adding new ones or altering existing ones is easier compared to having a single, large model.

In order to employ simulations in a hybrid system tool, our observers need to be translated into the modelling language of the tool. We showed an example translation, along with a general translation recipe, for the 20sim tool. Our methods for 20sim can be adapted for similar tools such as Simulink. The modelling of observers and simulations in a hybrid system tool is open to difficulties caused by syntactic and semantical differences between the hybrid automata formalism, and the modelling language and simulation engine of the hybrid system tool. However, we think that having a tool independent and precise formalism balances this drawback.

Apart from being able to employ simulations, another advantage of having testable observers is that one can easily combine observers via Boolean operations by simply applying associated Boolean operations on their outputs. We employed a four-valued logic to combine the results of individual observers. The four-valued logic we used is a Boolean logic extended with two auxiliary indeterminate values indicating the test result if the simulation is run to infinity and results of all the involved observers stay the same. We believe that using such a logic partly alleviates the inescapable limitations of using simulation on infinite traces. Overall demonstration of representing testable hybrid observers in a hybrid system simulation language and describing how to interpret simulation results is another contribution of this chapter.

Lastly, in this chapter, we adapted the validity criteria for contract-based specifications and the methods for validation introduced in Subsection 4.3.2 for the presented concrete A/G contracts formalism and our observer-based approach.

This chapter laid out the concrete and practical A/G contracts formalism to specify components with hybrid behaviour. In Chapters 6, 7 and 8, we will employ this formalism to specify the system components that form the basis of our specification workflow Subsection 4.3.1. We will either employ the concrete A/G contracts formalism with general hybrid behaviours as defined in this chapter or a restricted version of it with timed behaviours.

6

Specification of Mode Behaviour

In this chapter we define the steps for *formalization and validation of mode contracts* and *composition of mode contracts into a mode-composition contract*. These steps are depicted in Figure 6.1, which is the branch of the overall workflow shown in Figure 4.8 for specifying the mode behaviour of the system by a number of modes.

The aims of this chapter are, for observer-based mode and mode-composition contracts:

- Define and demonstrate the generic steps of contract formulation
- Define the specific validity criteria
- Demonstrate the methods for assessment of the validity criteria

In Section 6.1, we explain how individual mode contracts are formulated and validated for requirements given for a set of modes. During construction of individual mode contracts from the associated mode requirements, we explain how the requirement items are classified into assumptions and guarantees, and formalized as assertions using hybrid observers we introduced in Chapter 5. We also state and demonstrate how to validate individual mode contracts by simulating observers.

In Section 6.2, we explain how to compose a set of mode contracts into a single mode-composition contract. We state how the validity of the mode-composition contract can be derived from the validity of the constituent mode contracts. Demonstrations in both sections are performed on the robot case described in Chapter 3.

Figure 6.1: Mode behaviour specification workflow.

6.1 Specification of Mode Components

In this section we focus on specifying behaviour of a mode component with a mode contract. First, in Subsection 6.1.1, we present a generic mode component by defining its set of ports. Then, in Subsection 6.1.2, to be able to work on the robot case, we state the modes of the robot and present the initial requirements of a selected mode. Then, in Subsection 6.1.3, we demonstrate on the robot how to construct a mode contract by formalizing its requirements as assertions using the contract formalism introduced in Chapter 5. Lastly, in Subsection 6.1.4, we state and demonstrate how we can assess and improve the quality of a mode contract using the quality criteria presented in Section 5.5.

6.1.1 A Generic Mode Component

As seen in Figure 4.7, the mode-composition component is a composition of a number of mode components. A mode component interacts with the mode-switching component and the environment. In Figure 6.2, a generic mode component is depicted in the middle.

Figure 6.2: Generic mode component with its ports.

The set of ports of the generic mode component P_{mode} is the union of two disjoint sets: $P_{mode} = P^{sw} \cup P^e$. P^{sw} is the set of ports that are connected to the mode-switching component. P^{sw} includes a default input port assigned with activation and deactivation of the mode: $act_{mode} \in P^{sw}$. P^e is the set of ports associated to the variables of interest in the environment.

The ports in P^{sw} are discrete valued since they are connected to the mode-switching component, which is a discrete system. On the other hand, the ports in P^e can be continuous or discrete valued. A discrete port must have a finite domain, i.e. a finite set of values, that the discrete port can take whereas a continuous valued port has a continuous domain. For a particular instance of mode component, domains of individual ports can be further restricted.

As we stated in Subsection 5.3.2, a subset of ports of a component can be classified as inputs or outputs. For the generic mode component, the input and output ports are defined with two disjoint sets P^{in} and P^{out} such that $P_{mode} \supseteq P^{in} \cup P^{out}$. Since the ports in P^{sw} are connected only to discrete systems, each of them can be classified as an input or an output: $P^{in} \cup P^{out} \supseteq P^{sw}$. This argument does not apply to P^e . Controllability (that a port is controlled or uncontrolled by the mode component) of the ports in P^e can be complex. Environment variables are usually both constrained by the environment and by the mode component. The set of desired behaviours on an environment variable is dictated by the mode component. On the other hand, the set of all possible behaviours on the environment variable is defined by the environment, i.e. the laws of physics, the plant design, etc. Thus, it might not be possible to classify a port in P^e associated to an environment variable solely as an output (controllable) or an input (uncontrollable).

6.1.2 Case: Modes of the Robot

In this subsection, we first state the modes that are identified in our robot case. Then, we provide the initial requirements of one of the modes, the line following mode. Then, we define and classify the ports of the mode according to the generic mode component definition given in Subsection 6.1.1.

In our case study introduced in Chapter 3, the robot has five modes of operation shown in Figure 6.3. The functional descriptions of the modes are provided in Section 3.2. The mode m_2 , on which we demonstrate the construction of mode contracts,

is the line-follow mode. In this mode robot drives forward following the line. The requirements for m_2 are listed in Table 6.1.

Figure 6.3: mode-composition component is decomposed into five modes.

- R1. Initially, front border of the robot is located on the line, crossing it.
- R2. Initially, the speed of the robot is zero.
- R3. When activated, robot starts driving, following the line. The speed of the robot t_f time units after activation should be always within the range $v_d \pm l$.
- R4. The labyrinth has a predefined structure in terms of lines, junctions and places for obstacles. (See Chapter 3 for description of the labyrinth structure.)
- R5. While driving, if the robot bumps into an obstacle or faces a junction then the robot reports this on the port *ev_rep* by setting its value to *B* or *J*, respectively. The *ev_rep* should be set within t_d time units after the corresponding event. The *ev_rep* should stay set for a minimum of t_e time units and as long as the event signal (*bump* or *junction*) is *true*. After t_e time units elapsed and the event signal goes to *false* the *ev_rep* should be set to its normal value within t_d time units.
- R6. Robot has a predefined body and two wheels with a differential drive. (See Chapter 3 for description of the robot body.)
- R7. While driving, if the robot bumps into an obstacle, its velocity becomes zero within t_b time units.

Table 6.1: Textual requirements of the m_2 mode.

Figure 6.4: Ports between the m_2 component, and mode-switching component and environment.

According to the mode requirements, the interactions between the mode m_2 component, and mode-switching component and environment are depicted in Figure 6.4. The set of ports of the m_2 , P_{m_2} , is the union of two disjoint sets:

$$P_{m_2} = P_{m_2}^{sw} \cup P_{m_2}^e$$

where

$$P_{m_2}^{sw} = \{act, ev_rep\}$$

$$P_{m_2}^e = \{bump, junction, online, offline, v, p, \theta, w_l, w_r\}$$

The classification of the ports in $P_{m_2}^{sw} \cup P_{m_2}^e$ as input and output ports is as follows:

$$P_{m_2}^{in} = \{act\}$$

$$P_{m_2}^{out} = \{ev_rep\}$$

Meanings of the ports between the mode m_2 and the mode-switching components are as follows:

- *act*: The port that is for activation and deactivation of the mode by the mode-switching component.
- *ev_rep*: The port that is for the mode to report events to the mode-switching component. The port value can be *N*, *B* or *J* indicating that no event has occurred, that the robot has bumped into an obstacle and that the robot has faced a junction respectively.

Meanings of the environment variables used in the specification of m_2 mode are as follows:

- *bump*: State of the robot's touch sensor detecting an obstacle.
- *junction*: State of the robot's front line sensors being on a junction.
- *online*: State of the robot's front line sensors in the middle being on the line.
- *offline*: State of none of the robot's sensors detecting any line.
- *v*: Actual velocity vector of the center of rotation of the robot.
- *p*: Actual position vector of center of rotation of the robot.
- *θ*: Actual angle in radians between the robots driving direction vector and the positive x-axis.

- w_l and w_r : Scalar speeds of the left and right wheels.

The domains of the ports and variables are as follows:

- $D(act) = \{ON, OFF\}$
- $D(ev_rep) = \{N, B, J\}$
- $D(bump) = D(junction) = D(online) = D(offline) = \{true, false\}$
- $D(p) = D(v) = \mathbb{R}^2$
- $D(\theta) = D(w_l) = D(w_r) = \mathbb{R}$

Though we present the ports for the m_2 mode beforehand, some of the ports of a mode can only be defined while the requirements are formulated as assertions on variables and ports. This is especially the case for the ports in P^e . The requirements can be in natural language and/or they can be in a form not explicitly referring to the related environment variables. The ports in P^{sw} might be given beforehand or more easily deduced from the requirements before any formalization.

6.1.3 Case: Contract for the Line-following (m_2) Mode

In the previous subsection we presented the modes in the robot case and listed the initial requirements of the line following m_2 mode of the robot. In this subsection, we demonstrate how to construct a mode contract using the m_2 mode. In general, as the first step of constructing a mode contract, each requirement of the mode is classified as an assumption or a guarantee. Then, each requirement is formalized using observer-based assertions and the mode contract is formulated.

Recall that assumption assertions of a component contract are the responsibilities that the environment of the specified component has to fulfil whereas the guarantee assertions of a contract are the responsibilities of the component itself (Section 2.2). Please focus on the requirements for the mode m_2 presented in Table 6.1. Note that the requirements $R1, R2, R4, R6$ and $R7$ are the responsibilities of the environment, be it the other modes or the robot plant or the labyrinth. We classify these requirements as the assumptions of the m_2 mode. The assumption assertions A_1, A_2, A_3, A_4 and A_5 refer to the requirements $R1, R2, R4, R6$ and $R7$, respectively. On the other hand, the requirements $R3$ and $R5$ are the responsibilities that the mode component should fulfil when placed in an appropriate environment. We classify these requirements as the guarantees of the m_2 mode. The guarantee assertions G_1 and G_2 refer to the requirements $R3$ and $R5$, respectively. This mapping between the requirements and assumption and guarantee assertions is summarized in Table 6.2.

Assumptions	
A_1	$R1$
A_2	$R2$
A_3	$R4$
A_4	$R6$
A_5	$R7$

Guarantees	
G_1	$R3$
G_2	$R5$

Table 6.2: Classification of the requirements of the mode m_2 as assumptions and guarantees.

After the requirements of a mode are classified into assumptions and guarantees, each requirement can be formalized as an hybrid observer and the mode contract can be constructed. A mode contract $C_M = (A_M, G_M)$ for a mode M is an assume/guarantee contract where A_M and G_M represent the overall assumption and guarantee assertions. For the mode m_2 , the A/G-contract is as follows:

$$C_{m_2} = (A_{m_2}, G_{m_2})$$

Since we directly associated each requirement of m_2 to an assertion (See Table 6.2), the overall assertions A_{m_2} and G_{m_2} are conjunctions of the assumption and guarantee assertions that correspond to the requirements of m_2 :

$$\begin{aligned} A_{m_2} &= A_1 \wedge A_2 \wedge A_3 \wedge A_4 \wedge A_5 \\ G_{m_2} &= G_1 \wedge G_2 \end{aligned}$$

According to Subsection 2.2.2, in order apply the contract algebra, i.e. calculate contract composition according to Equation (2.1), we also need to make C_{m_2} saturated. Recall that an A/G contract $C = (A, G)$ is saturated if and only if $G \supseteq \neg A$. To make C_{m_2} saturated we need to revise the guarantee assertion as follows:

$$G_{m_2} = (G_1 \wedge G_2) \vee \neg A_{m_2}$$

Note that the defined assertions have usually different alphabets. When we compose assertions we always assume that their alphabets are equalized by the alphabet equalization operator defined in Section 5.1.

In Section 2.2, we stated that using a formal language to express to contracts is essential due to the exactness and executability of formal contracts. Later, in Chapter 5, we introduced hybrid observers to express formal A/G contracts for system specification. In order to obtain a formal contract for a mode, the mode requirements need to be formalized using hybrid observers. Here, we demonstrate the formalization process on the mode m_2 of the robot. The observers for the assumption assertions of m_2 can be seen in Figure 6.5, Figure 6.6 and Figure 6.7. The observers for the guarantee assertions of m_2 can be seen in Figure 6.8 and Figure 6.9.

In Figures 6.5-6.9, some of the behaviours that are common in the observer models for any mode are omitted for the sake of clarity. In all of the observer models for a mode, there is only one initial location where the observer is always waiting for the activation, i.e. the variable *act* to become *ON*. Only after activation, the observer starts observing its input ports and goes to a bad location if the corresponding assertion is violated. When an observer goes to a bad location, it stays there forever. At any location except a bad location, if *act* becomes *OFF*, then the observer directly goes to the initial location. That is, in an observer, every deactivation event coming after an activation event, takes the current location, except the bad location, to the initial location. In this sense, a mode is historyless between the activation periods, with the exception of the bad state. These transitions for *act* being false are omitted from observer diagrams.

Also, according to Section 5.3, at any location the guards of the outgoing transitions must be disjoint.

In Figure 6.5, an observer for the intersection of A_1 and A_2 , i.e. $A_1 \wedge A_2$, is shown. In Figure 6.6 the observer automaton for A_3 is shown. *Line*, *Obs* and *Junction* functions in Figure 6.6 are look-up functions with:

$$Line: \mathbb{R}^3 \mapsto \{true, false\}$$

$$Obs: \mathbb{R}^3 \mapsto \{true, false\}$$

$$Junction: \mathbb{R}^3 \mapsto \{true, false\}$$

such that they show potential places for lines, obstacles and junctions for a properly structured labyrinth as described in Section 3.2.

In Figure 6.7, the observer for $A_4 \wedge A_5$ is shown. A_4 and A_5 describe assumptions on robot's dynamic behaviour without and with a bump into an obstacle. Therefore we formalized them together with a single observer in order to prevent any conflicts. The *Pos* and *Ang* functions in Figure 6.7 encode the kinematics of the two-wheeled differential drive robot as described in Section 3.1. In the larger location in the middle, these functions compute the values of the reference variables θ_t and p_t for orientation and position of the robot using the wheel speed variables, w_l and w_r . It goes to a bad state if the values of the variables, θ and p , for actual orientation and the position of the robot do not match the values of the reference variables, or the actual velocity does not match the derivative of reference position. In case of a bump, i.e. if the *bump* port becomes *true*, the speed of the robot needs to become zero within t_b time. Otherwise the automaton goes to a bad state.

Figure 6.5: Observer automaton for $A_1 \wedge A_2$.

Figure 6.6: Observer automaton for A_3 .

Figure 6.7: Observer automaton for $A_4 \wedge A_5$.

The observer for G_1 is shown in Figure 6.8. The observer goes to a bad state if, within t_f time units after activation, the robot is not online or the robot's speed $|v|$ is not within $v_d \pm l$.

The observer for G_2 is shown in Figure 6.9. Though the G_2 observer looks complex, the top and bottom half of it are the same, each formulating the behaviour described in R_3 of Table 6.1 for bumping into an obstacle and facing a junction. We explain only the part for bumping into an obstacle ignoring the bottom three locations and the transition involving those. Upon activation the observer goes to a location in which it should not set the value of ev_rep to B , which is reserved for reporting a bump. If any time a bump is detected, i.e. $bump$ becomes true, the observer goes to a location where the ev_rep must be set to B within t_d time units. When ev_rep is set to B , the observer goes to a location where ev_rep stays set to B until $bump$ goes to false and t_c time units passed. A lower bound of t_c time units is used to make signalling the bump on ev_rep for a minimum period. After $bump$ gets false and t_c time units is passed the ev_rep must be set to a value that is not B within t_d time units.

Figure 6.8: Observer automaton for G_1 .

Note that for the assertions A_3 , G_1 and G_2 we formulated individual observers. On the other hand for the assertions A_1 , A_2 , A_4 and A_5 we formulated two observers, combining A_1 and A_2 , and A_4 and A_5 . In the case of $A_1 \wedge A_2$, our motivation is convenience; both A_1 and A_2 asserts a condition at time 0. In the case of $A_4 \wedge A_5$, our motivation is to prevent any conflicts; A_4 and A_5 describe assumptions on robot's dynamic be-

Figure 6.9: Observer automata for G_2 .

haviour without and with a bump into an obstacle. In general, the choice of separately expressing assertions or combining them is guided by experience.

6.1.4 Validity of Mode Contracts

In this subsection, we state and demonstrate how we can employ the three validity criteria defined in Section 5.5 to assess the validity of a mode contract. For each criterion:

- We give informal/formal definition of the validity criterion for a mode contract.
- We provide methods and important points for assessing and improving the criterion on a mode contract.
- We demonstrate these methods and points on the robot case.

Completeness

According to the definition of completeness given in Subsection 4.3.2, a mode contract is called *complete* if it defines all the necessary properties of the envisioned mode. Completeness is actually a property of the initial mode requirements. It asks if the requirements listed for a mode are complete. We denote completeness of a contract for a mode M as follows:

$$\kappa(C_M)$$

As stated in Subsection 4.3.2 and in Section 5.5, there is no formal definition of this criterion since, when defining the initial requirements, there is no formal reference against which it can be checked. However, given that the mode contract is correct (see the next point on correctness) with respect to the mode requirements, one can execute and analyse the formalized requirements, i.e. assertions of the mode contract, to

explore the completeness of the specification. Completeness of a mode contract M implies completeness of the associated mode requirements given that the mode contract is correct:

$$C_M \leq R_M \implies (\kappa(C_M) \Rightarrow \kappa(R_M))$$

In order to assess completeness of a mode contract we need to investigate various behaviours of a system described by the contract. For this, we can use simulation if the contract is observer based. As described in Subsection 5.4.2, the way to use observers in simulation is to generate traces and feed them to the observers. A question is how to generate such traces in order to properly assess the completeness of a contract. An approach is to use early abstract system models.

Focusing on the mode m_2 , in order to investigate the behaviours of m_2 specified by the contract C_{m_2} , we can use a model for m_2 as shown in Figure 6.10. Here, as the trace source for m_2 , an early design model for m_2 for driving straight is used. The model for m_2 is composed of components for an activation unit for m_2 , a controller for m_2 , the plant model for the robot and a maze model. The traces generated by this model is fed into the observers for G_1 and A_{m_2} . Note that, this early design model does not constrain the design; it is for exploring the validity of specification. In the design process, a completely different design can be employed.

Figure 6.10: 20-sim block diagram for simulation of A_{m_2} and G_1 .

After simulating the model in Figure 6.10, we notice an incompleteness. In Figure 6.11 a collection of plots of the simulation of a sample trace is shown. The simulated trace is accepted by A_{m_2} during the whole simulation; the output of A_{m_2} , i.e. A_m2_out , is always true. But the same trace is not accepted by G_1 ; the output of G_1 , i.e. $G1_out$, goes to false at $t = 4s$. This shows that the simulated trace invalidates $A_{m_2} \Rightarrow G_1$, hence it is not a valid trace for an implementation of C_{m_2} . This is not a problem as long as it is intended. However, as we will see, the simulated trace should be a valid trace and the result we see in Figure 6.11 is due to an incompleteness regarding the assertions $A_4 \wedge A_5$ and G_1 .

Figure 6.11: Simulation of A_{m_2} and G_1 with a behaviour including a bump.

Note that $A_4 \wedge A_5$ (Figure 6.7) assumes that the robot to come an halt after it is blocked by an obstacle. However, the observer for G_1 (Figure 6.8) might go into a bad state in such a case since robot's speed will be zero due to the bump. The top four plots in Figure 6.11 depict the ports *act*, *speed*, *bump*, *online*, where *speed* = $\|v\|$. The bottom two plots depict the values of the output values of the observers A_{m_2} and G_1 . The plots depict the simulation of a sample trace in which the mode is activated at $t = 0.5s$ and a bump occurs at $t = 4s$. Through the whole time the robot is online. The speed of the robot goes up from 0 to about 0.19 m/s after activation within 2 seconds, which conforms to G_1 . At the point the robot bumps into an obstacle, the robot comes to an halt conforming to $A_4 \wedge A_5$. However, the bump causes an invalidation of G_1 , which requires that the robot should maintain a certain speed after activation. This shows that any trace that is included in the assertion defined by $A_4 \wedge A_5$ that includes a bump within the activation period will violate $A_5 \Rightarrow G_1$. Since such trajectories also conform to the whole assumption assertion A_{m_2} , they will also violate $A_{m_2} \Rightarrow G_1$. This is not intended and due to the incompleteness of the requirements, specifically R_3 (Table 6.1). The requirement that the robot should drive on the line should become void when the robot bumps into an obstacle. Thus, we need to update R_3 and the corresponding

observer G_1 . R_3 is replaced by: R_3 . When activated, robot starts driving, following the line. The speed of the robot t_f time units after activation should be always within the range $v_d \pm l$ unless the robot bumps into an obstacle.

The observer G_1 is modified as in Figure 6.12. Note that the guard for the transition from the middle location to the rightmost one now requires that there is no bump, $\neg bump$.

Figure 6.12: Observer automaton for G_1 , modified for completeness.

Correctness

By ‘correctness of a mode contract’, we mean informally that *a mode contract refines a list of requirements given for that mode*. That is, a contract C_M for a mode M with requirements R_M is correct if and only if:

$$C_M \leq R_M$$

The requirements for correctness of a mode contract are:

1. Correct identification of the system ports that are related to the mode function.
2. Correct classification of the requirements as assumptions or guarantees.
3. Correct formalization of the individual requirements as formal assertions.

There is no formal definition of correctness in this context since informal textual requirements are involved. However, by executing and analysing the individual assertions we can assess their correctness with respect to the associated requirements.

Related to the third item above, one needs to ensure that assertions and guarantees do not constrain outputs and inputs of the mode component, respectively. We should check that assumption observers do not constrain the output variables in $P_{m_2}^{out}$ and similarly that guarantee observers do not constrain the input variables in $P_{m_2}^{in}$. Both assumptions and guarantee assertions can constrain the variables that are defined neither as inputs or outputs in P^e . Formally, in an hybrid observer, guards of the transitions going to a bad state cannot include the variables that should not be constrained by the observer. The observers constructed for the m_2 mode conform to this criterion.

Consistency

According to Subsection 5.5.3 we stated the condition for consistency of a given saturated A/G contract $C = (A, G)$:

- C defines consistent implementations and legal environments, i.e. $B = A \wedge G \neq \emptyset$

However, for a mode contract to be consistent, we need stronger consistency criteria than the ones stated above. If we use the generic consistency criteria as they are on the modes formalized with historyless observers, any mode contract would be consistent since it would at least accept the trajectories that never activates the mode, i.e. the trajectories that have the *act* port connected to the mode always *OFF*. Based on this observation, reformulation of the consistency criterion for mode contracts are as follows:

Definition 6.1.1. A saturated mode contract $C_M = (A_M, G_M)$ is consistent if and only if it satisfies the following condition:

- C_M defines consistent implementations and legal environments:

$$B_M = A_M \wedge G_M \neq \emptyset \quad \wedge \quad ON \in \text{RANGE}(pr_{act}(B_M))$$

□

The revised validity criterion in Definition 6.1.1 is suitable for checking validity of mode contracts since they require the trajectories in the assertions A_M , G_M and B_M to include activation of modes.

In order to validate that a mode contract C_M is consistent, it would suffice to show a witness trace for each of the three conditions stated in Definition 6.1.1. For instance to show that a mode contract $C_M = (A_M, G_M)$ satisfies the first condition of consistency on feasibility of legal environments, we need to find a hybrid trace χ such that:

$$\chi \in B_M \quad \wedge \quad ON \in pr_{act}(\chi)$$

In practice, again, we can employ simulations to perform this check. For instance, to show that the contract C_{m_2} for the m_2 mode of our robot case defined consistent legal environments and implementations, we can run simulations involving the observers for the individual assertions that make up the assumption A_{m_2} and guarantee G_{m_2} of C_{m_2} . To show that C_{m_2} is consistent, it is sufficient to find a trace simulation of which results in a true value on $A_{m_2_out} \wedge G_{m_2_out}$ as we did in Figure 6.11. We omit the associated simulations results here.

6.2 Specification of Mode-Composition Component

In this section, we focus on the specification of the mode-composition component, which is the composition of all the mode components. First, in Subsection 6.2.1, we present the architecture of the mode-composition component and give its contract formulation as the composition of the mode contracts. Then, in Subsection 6.2.2, we partially demonstrate constructing the mode-composition contract for the robot case, by composing two mode contracts. Lastly, in Subsection 6.2.3, we state how we define and assess the validity of a mode-composition contract.

6.2.1 Mode-Composition Component Architecture

In this subsection, we state the architecture of the mode-composition component and give the contract formulation as composition of the mode contracts. We explain the relationship between the ports of the mode components and define the composability on the ports of the modes.

In Figure 4.7, we showed the decomposition of the mode-composition component into a number of mode components. Following this decomposition, the generic contract formulation for the mode-composition component is defined as follows:

$$C_{mc} = \bigotimes_{i \in M} C_{m_i} \quad (6.3)$$

where M denotes the set of indices for the modes of the system to be specified. For the robot case, the mode-composition contract C_{mc} is defined as follows (See Figure 6.3):

$$C_{mc} = \bigotimes_{i \in [1,5]} C_{m_i}$$

Ports of the Mode Components and Composability.

Regarding the composability, any two modes can not share output ports, since an output port can only be controlled by a single mode component. Also, recall that the modes do not directly interact with each other (Subsection 6.1.1). One's input port cannot be connected to another's output port. These two constraints require that a mode component's output port cannot be included in another mode component's ports:

$$\forall i, j \in M \wedge i \neq j : P_{m_i} \cap P_{m_j}^{out} = \emptyset$$

Modes, however, may interact with each other through the environment, either being active at the same time or being activated successively. Two modes might have requirements, as assumptions or guarantees, defined on the same environment variables.

Each mode component have an activation port which is used to activate and deactivate the mode by the mode-switching component. For a mode component m_i , we name the associated activation port as act_{m_i} . The ports of the mode-composition component is the union of the ports of the mode components since there are no eliminated ports during the composition:

$$P_{mc} = \bigcup_{i \in M} P_{m_i}$$

6.2.2 Case: Contract for the Stopping (m_1) Mode

In this subsection, we present the contract for the stopping m_1 mode from the robot case (Chapter 3) and give the formulation of composition of two mode contracts. We present the requirements of the mode m_1 in Table 6.3

- R1.* Within t_s units of time after activation, the robot's speed should become and stay zero.
- R2.* Robot has a predefined body and two wheels with differential drive. (See Chapter 3 for description of the robot body.)

Table 6.3: Textual requirements of the m_1 mode.

Similar to what we did for m_2 in Subsection 6.1.3, we first need to classify the requirements of m_1 , and then to formalize them using hybrid observers. The item *R1* in Table 6.3 is a guarantee, and *R2* is an assumption. We call the assertion for *R1* as G_1 , and *R3* as A_1 . The observer for G_1 is shown in Figure 6.13. In Figure 6.13, within t_s time after activation, the robot is required to stop. The observer for A_1 is the same as the observer formulated for the mode m_2 shown in Figure 6.7.

Figure 6.13: Observer automaton for the assertion G_1 .

The saturated contract C_{m_1} for m_1 is as follows:

$$C_{m_1} = (A_{m_1}, G_{m_1}) = (A_1, (G_1 \vee \neg A_1))$$

Now, we can demonstrate how to construct a mode-composition contract by composing the individual mode contracts. We partially construct the mode-composition contract for the robot case by composing m_1 and m_2 as follows:

$$C_{m_1 m_2} = C_{m_1} \otimes C_{m_2} \quad (6.4)$$

Since C_{m_1} and C_{m_2} are saturated contracts, we can formulate the composition as follows:

$$C_{m_1 m_2} = ((A_{m_1} \wedge A_{m_2}) \vee \neg(G_{m_1} \wedge G_{m_2}), G_{m_1} \wedge G_{m_2})$$

Note that the composition, $C_{m_1 m_2}$, is also saturated.

6.2.3 Validity of Mode-Composition Contract

In this subsection, we focus on the validity criteria for the mode-composition component. We state the relationship between the validity of the mode-composition contract and individual mode contracts. For the mode-composition contract, we have the same three validity criteria defined in Section 5.5: correctness, completeness and consistency.

Correctness

The correctness of mode-composition contract is defined with respect to the requirements of the individual modes. The mode-composition contract is correct if and only if:

$$C_{mc} = \bigotimes_{i \in M} C_{m_i} \leq \bigotimes_{i \in M} R_{m_i}$$

By extending the independent implementability (Subsection 2.2.1) to requirements, correctness of the mode-composition contract is implied by the correctness of the mode contracts:

$$\bigwedge_{i \in M} (C_{m_i} \leq R_{m_i}) \implies C_{mc} \leq \bigotimes_{i \in M} R_{m_i} \quad (6.5)$$

Completeness

Since while constructing C_{mc} we only compose the mode contracts and there is no specification defining the behaviours associated with activation and deactivation of the individual modes, completeness of the mode-composition contract C_{mc} , i.e. $\kappa(C_{mc})$, is also implied by the completeness of the individual mode contracts C_{m_i} :

$$\bigwedge_{i \in M} \kappa(C_{m_i}) \implies \kappa(C_{mc}) \quad (6.6)$$

Consistency

The mode contracts included in a mode-composition contract can be inconsistent if they declare inconsistent behaviours in their common ports. The mode contracts do not have common output ports. However, they can have ports to the same plant variables on which they can declare assumptions and guarantees. They can also share input ports. So, the composition of the mode contracts, that is the mode-composition contract C_{mc} , can be inconsistent since the modes might define different values for the shared ports. However, again, the fact that the mode-composition contract C_{mc} does not define any behaviour regarding the activation and deactivation of individual modes makes C_{mc} a consistent contract.

In order to illustrate this, let C_i be a parametrized contracts such that

$$\begin{aligned} C_i &= (A_i, G_i) \\ P_{C_i} &= \{act_i, out\} \\ A_i &= True \\ G_i &= \text{"If } act_i = True, out \text{ takes the value } i\text{"} \end{aligned}$$

The two contracts generated from C_i for $i = 1, 2$ clearly have inconsistent guarantee assertions for the durations they are both active. However, the composition

$$C_1 \otimes C_2 = (True, G_1 \wedge G_2)$$

is consistent. $G_1 \wedge G_2$ is non-empty: it consists of the behaviours which (1) satisfy G_1 and G_2 and (2) do not have both C_1 and C_2 active at the same time.

6.3 Conclusions

In this chapter we showed how to specify modes of a system by formulating mode contracts and then composing them into a mode-composition contract. In Section 6.1, we have stated that a mode contract is constructed by classifying and formalizing the individual requirements of the mode. While expressing the mode contracts, the use of the concrete A/G contracts formalism with hybrid observers introduced in Chapter 5 enhances the process of formalization and validation. To be able to combine the hybrid observers effortlessly enables formalizing the individual mode requirements and easily composing them into contracts, hence reducing the complexity of the process of formalization of the requirements. Also, being able to simulate the hybrid observers, we can utilize hybrid system simulation tools in the assessment of validity of mode contracts.

While working on a mode of the robot case, we have seen that assessing validity of mode contracts requires modifications of the general A/G contracts validity criteria defined in Subsection 4.3.2. This shows that an effective and useful definition of validity is different for the components in our specification architecture.

In Section 6.2, we have shown how to compose mode contracts of a specification into the mode-composition contract. We have shown that composition of mode contracts into the mode-composition contract is a straightforward process and the validity of mode-composition contract is implied by the validity of individual mode contracts. The more involved validation actions will come when the switching of the modes is involved while the overall system contract is constructed in Chapter 8.

7

Specification of Mode Switching

In this chapter we explain how to construct and validate a mode-switching contract indicated in the specification workflow in Section 4.3. First, in Section 7.1, we define a generic mode-switching component for our specification architecture by defining a generic set of ports and properties. In Section 7.2, we show that the assertions of mode-switching contract can be defined as sets of restricted hybrid behaviours, namely timed behaviours. We explain how we can employ the Uppaal, a tool to express and analyse such assertions. Then, in Section 7.3, we explain and demonstrate how we to construct the switching contract from an initial set of informal requirements. Lastly, in Section 7.4, we show how to assess validity of the switching contract using the simulation and formal verification in Uppaal tool.

7.1 Generic Mode-Switching Contract

In Section 4.3, we explained that our specification approach is based on a mode-based specification architecture which involves a number of mode components and a mode-switching component. In this direction, we assumed that the initial system requirements are structured/grouped into a number of modes and a mode switching logic. The mode-switching contract formalizes the system requirements related to mode switching logic.

The mode-switching component is responsible for determining the active modes of the system based on the system's interaction with the other systems and the environment (See Figure 4.6). It directly interacts with other systems, but it does not directly interact with environment. Therefore, the mode-switching contract does not refer to any environment variables. As dictated by the specification architecture in Section 4.3, we delegate dealing with the environment variables to the mode components. If needed, one can define a mode that just measures an environment variable and reports it to the switching component.

Figure 7.1: Ports of the generic mode-switching component.

In Figure 7.1, ports of the generic mode-switching component is shown. Ports of the switching component are defined by the union of two disjoint port sets:

$$P_{sw} = P_{sw}^{os} \cup P_{sw}^{mc}$$

where P_{sw}^{os} denotes the set of ports from and to the other systems, and P_{sw}^{mc} denotes the set of ports from and to the mode-composition component.

For each mode component, there is an associated activation port in P_{sw}^{mc} :

$$P_{sw}^{mc} \supseteq \{act_\chi \mid \chi \text{ is a mode component}\}$$

All the ports in P_{sw} have discrete domains and they are directed, i.e. a port is either an input or an output port:

$$P_{sw} = P_{sw}^{in} \cup P_{sw}^{out} \quad \text{where} \\ P_{sw}^{in} \cap P_{sw}^{out} = \emptyset \quad \text{and} \\ P_{sw}^{out} \supseteq \{act_\chi \mid \chi \text{ is a mode component}\}$$

7.2 Assertions for Mode Switching

In this section, we explain the formalism for the assertions used to specify switching behaviour and the methods to analyse the assertions. First, we describe the sort of behaviours that we need to capture in a switching contract, namely timed trajectories. Then, we will present timed automata formalism, which is a subset of hybrid automata presented in Chapter 8, to formulate switching assertions. Then, we will describe the use of the Uppaal tool (Behrmann et al., 2006) to perform simulations and verifications of the assertions formulated as timed automata.

7.2.1 Timed Automata and Timed Observers

In Chapter 6, we constructed hybrid observers to formalize the individual requirement items for a mode's behaviour as assertions. Then, we combined these observer-based assertions in the mode contract which is an A/G contract. We can also use a similar approach to formulate the switching contract. However, for the reasons below, it is not ideal.

First, as we will shortly explain, that the ports of the switching component being discrete valued allows us to employ timed-automata behaviours instead of general hybrid-automata behaviours, in specifying the switching behaviour. This fact opens up the possibility of using formal verification, which is not the case for general hybrid automata. In contrast to partial verification by simulation, with formal verification, we can fully verify if an abstract system model conforms to a set of observers or a set of properties.

Second, specification of a switching component tends to be more complex than of a mode component. In a system, a mode component is responsible of carrying out a specific and limited objective on the environment, such as keeping the speed of a car in a desired interval. Typically, as we demonstrated in Section 6.1, requirements for a mode can be expressed in a handful of requirement items each of which can then be formalized by not-so-complex observers. On the other hand, a mode-switching component is responsible of realizing the overall objective of the system by communicating with other systems and with the modes, and orchestrating the activation and deactivation of the modes. This indicates that the requirements for a switching component would be composed of many items. Consequently, if we follow a similar approach in constructing the switching contract, i.e. formalizing each individual requirement item by a separate observer, it would result in many observers. Assessing the validity of a contract composed of many observers would be very complex and error prone. To see this, let C_{sw} be a switching contract and M be an abstract model that models both the switching component and its environment. Also, let M and the observers in C_{sw} be in timed automata formalism. Let the sets of behaviours of M and C_{sw} be denoted by B_M and $B_{C_{sw}}$. Then we can employ a model checking tool, such as Uppaal, to formally check if M conforms or violates C_{sw} . That M conforms to C_{sw} shows that C_{sw} is consistent, i.e. the set of behaviours specified by it is not empty:

$$B_M \subseteq B_{C_{sw}} \wedge B_M \neq \emptyset \Rightarrow B_{C_{sw}} \neq \emptyset$$

In order to see if C_{sw} is sufficiently complete for our purposes we can investigate M further via simulation and/or verification in Uppaal. However, even if we are satisfied that M is complete, we cannot infer the completeness of C_{sw} :

$$B_M \subseteq B_{C_{sw}} \wedge \kappa(B_M) \not\Rightarrow \kappa(B_{C_{sw}})$$

The reason is that $B_{C_{sw}}$ might include behaviours that are not in B_M and disallowed for the intended behaviour of the switching component. Note that, even though this argument on completeness also applies to assessment of validity of mode contracts, it is less effective due the two differences between mode and switching contracts stated above, i.e. that the mode specifications are less complex and that the simulation is the only option while working with mode contracts.

If we use non-observer models to express the mode switching specification, i.e. modelling the behaviour of the switching component as a timed automaton, then we can directly perform validity assessment on these models in Uppaal using formal verification and simulation. However, as stated in Section 5.3, it is not effective to check refinement or implementation relations between a contract that involves non-observer

assertions and component models or other observer-based contracts such as the mode-composition contract. Fortunately, timed-automata models can also be used as observers using an extension of Uppaal (Larsen et al., 2004).

We can express all or parts of a switching contract using non-observer assertions and use formal verification and simulation in Uppaal to assess the validity of the switching contract. Then, when we need to, we can use this switching contract as an observer-based contract. For instance, in assessing the validity of the overall system contract, which is the composition of the mode switching and the mode-composition contracts, we need to simulate the observer-based mode contracts and non-observer switching contract together. As we will see in Chapter 8, being able to use a switching contract composed of non-observer assertions in Uppaal as an observer enables simulating the whole overall system contract in 20-sim.

Now, we will explain the sort of behaviours we need for mode switching specification and the formalisms to express the set of behaviours. A mode-switching contract, C_{sw} , is an A/G-contract composed of a pair of assume and guarantee assertions:

$$C_{sw} = (A_{sw}, G_{sw})$$

Like before, the assertions A_{sw} and G_{sw} are logical combinations of atomic assertions. Any behaviour of a switching component can be expressed as a hybrid trajectory (Definition 5.1.4) defined over the ports of the switching component and an assertion is a set of hybrid trajectories.

However, note that all the ports of the mode-switching component have discrete domains, and the only entity that has a continuous domain is time. For this reason, in specifying the mode switching behaviour, it is sufficient to use assertions defined by timed automata (Alur and Dill, 1994) instead of hybrid automata. Before defining timed automata, we define the notion of clock constraints which we will use in the definition of timed automata. The following definition is taken from (Lygeros et al., 2008).

Definition 7.2.1. Let $X = \{x_1, \dots, x_n\}$ be a finite collection of variables with $D(x_i) = \mathbb{R}$ for all i . Let also that $x = (x_1, \dots, x_n) \in \mathbb{R}^n = \mathbf{X}$ denote a valuation for all $x_i \in \mathbf{X}$. Then, the set, $\Phi(X)$, of *clock constraints* of X , is a set of logical expressions defined inductively by $\delta \in \Phi X$ if:

$$\delta := (x_i \leq c) \mid (x_i \geq c) \mid \neg \delta \mid \delta_1 \wedge \delta_2$$

where $x_i \in X$ and $c \geq 0$ is a rational number. With each $\delta \in \Phi(X)$, we can associate a set:

$$\hat{\delta} = \{x \in \mathbf{X} : \delta(x) = \text{True}\}$$

□

Timed automata is a restricted version of hybrid automata (Definition 5.2.1). The following definition is taken from (Lygeros et al., 2008).

Definition 7.2.2. A *timed automaton* $TA = (Q, X, \text{Init}, f, \text{Inv}, E, G, R)$ is a hybrid automaton, where

- Q is a finite set of discrete variables, $\mathbf{Q}$ is finite and called the set of locations, $\mathbf{Q} = \{q_1, \dots, q_m\}$;
- X is a finite set of continuous clock variables, $\mathbf{X} = \mathbb{R}^n$;
- $Init = \{\{q_i\} \times \widehat{Init_{q_i}}\}_{i=1}^m$ where $Init_{q_i} \in \Phi(X)$;
- $f(q, x) = (1, \dots, 1)$ for all (q, x) ;
- $Inv(q) = \widehat{Inv_q}$ where $Inv_q \in \Phi(X)$ assigns to each $q \in \mathbf{Q}$ an invariant set;
- $E \subset \mathbf{Q} \times \mathbf{Q}$ is a collection of discrete transitions;
- $G(e) = \widehat{G_e}$ where $G_e \in \Phi(X)$, for all $e = (q, q') \in E$; and
- For all e , $R(e, x)$ either leaves x_i unaffected or resets it to 0.

□

The main difference between a timed automaton and a general hybrid automaton is that the continuous variables in timed automaton are limited to variables that keep the passage of time, i.e. the clocks. In this direction, in a timed automaton, all of the definitions relating to clocks are restricted. First, continuous flow of a clock variable c_i is always defined by $\dot{c}_i = 1$ for every location of the automaton. Second, predicates defined on clocks while defining initial states, location invariants and edge guards of an automaton is limited to clock constraints defined in Definition 7.2.1. Third, in a reset relation of an edge, a clock can only be reset to 0 or left at the same value. All of these restrictions enable the formal verification of certain real-time properties of a timed automaton to be decidable (Alur and Dill, 1994).

Since a timed automaton is still a hybrid automaton, the formal definition of its executions are defined similarly (Definition 5.2.2).

Using timed automata, we can express assertions for components of which behaviours are timed trajectories of a set of discrete variables. For a component M with an alphabet P of discrete variables, we can use a time automaton $TA = (Q, X, Init, f, Inv, E, G, R)$ to express the assertions for M . Then the trajectories over the alphabet P of M contained in the assertion expressed by TA are the set of executions of TA equalized to P , i.e. $pr_P^{\sim}(Exec(TA))$. We denote the assertion specified by a timed automaton TA on an alphabet P by $Asrt_P(TA) = pr_P^{\sim}(Exec(TA))$.

A timed automaton defined according to Definition 7.2.2 is closed in the sense that it does not have inputs and outputs. Since in the switching component each of the ports are classified as an input or an output, an open timed automaton is more suitable to express assertions on switching, where there would be a natural mapping between the input and outputs ports of the specified component and the inputs and outputs of the automaton. An open timed automata is a restricted form of open hybrid automaton Definition 5.2.4, which is defined as follows:

Definition 7.2.3. An *open timed automaton* OTA is an open hybrid automaton $OTA = (Q, X, V, Y, Init, f, h, Inv, E, G, R)$, where

- Q is a finite set of discrete variables, $\mathbf{Q}$ is finite and called the set of locations, $\mathbf{Q} = \{q_1, \dots, q_m\}$;
- X is a finite set of continuous clock variables, $\mathbf{X} = \mathbb{R}^n$;
- V is a set of discrete input variables;

- Y is a set of discrete output variables;
- $Init = \{\{q_i\} \times \widehat{Init_{q_i}}\}_{i=1}^m$ where $Init_{q_i} \in \Phi(X)$;
- $f(q, x, v) = (1, \dots, 1)$ for all (q, x, v) ;
- $h: \mathbf{Q} \rightarrow \mathbf{Y}$ is a vector field;
- $Inv(q) = \{(x, v) : x \in \widehat{Inv_{q,x}} \wedge v \in Inv_{q,v}\}$ where $Inv_{q,x} \in \Phi(X)$ and $Inv_{q,v} \subseteq \mathbf{V}$ assigns to each $q \in \mathbf{Q}$ an invariant set;
- $E \subset \mathbf{Q} \times \mathbf{Q}$ is a collection of discrete transitions;
- $G(e) = \{(x, v) : x \in \widehat{G_{e,x}} \wedge v \in G_{e,v}\}$ where $G_{e,e} \in \Phi(X)$ and $G_{e,v} \subseteq \mathbf{V}$, for all $e = (q, q') \in E$; and
- For all $(e, v) \in E \times \mathbf{V}$, $R(e, x, v)$ either leaves x_i unaffected or resets it to 0.

□

The executions, traces, compatibility and composition for open timed automata are defined by Definition 5.2.5, Definition 5.2.6, Definition 5.2.7 and Definition 5.2.8, respectively. For a component M with alphabet $P = P_{in} \cup P_{out}$, an assertion can be expressed by an open timed automaton $OTA = (Q, X, V, Y, Init, f, h, Inv, E, G, R)$ with $V \subseteq P_{in}$ and $Y \subseteq P_{out}$ by taking projection of the traces of OTA , i.e. $Asrt_P(OTA) = pr_{\tilde{P}}(Trace(OTA))$.

In Section 5.3, we explained the difference between expressing an assertion with a hybrid automaton and a hybrid observer. For similar reasons stated there such as online testability, one might to express assertions with timed observers instead of timed automata. Using the open timed automata definition we can define a timed observer as follows:

Definition 7.2.4. A *timed observer* $TO = (Q, B, X, V, Y, Init, f, h, Inv, E, G, R)$ is an open timed automaton of which some locations are tagged as ‘bad’ locations defined by the set $B \subseteq \mathbf{Q}$. □

Note that an assertion expressed by a timed observer would again be limited to safety properties. In order for a timed observer TO to be online testable (see the discussion of online testability in Subsection 5.3.1), we need TO to be non-blocking and deterministic:

Definition 7.2.5. A *testable open observer* $TTO = (Q, B, X, V, Y, Init, f, h, Inv, E, G, R)$, is a timed observer which is non-blocking and deterministic. □

The sufficient conditions stated in Definition 5.3.9 for a hybrid observer to be testable also apply to timed observers. Note that the first item in Definition 5.3.9

1. For every location $q \in \mathbf{Q}$, $f(q, x, v)$ is Lipschitz continuous in x and piecewise continuous in v .

is always valid for a timed observer since, in a timed automaton, $f(q, x, v) = (1, \dots, 1)$ for all (q, x, v) .

As we will see in Section 8.2, a convenient feature of timed automata is that we can use a non-observer timed automaton as a timed observer by state space exploration in Uppaal.

7.2.2 Using Uppaal on Simulation and Verification of Switching Contract

In this subsection, first we introduce Uppaal (Behrmann et al., 2006), a tool for modelling, simulation and verification of real-time systems. Then, we will explain how one can use Uppaal to express, simulate and verify assertions of switching contract, which are formulated using timed automata.

Uppaal is a toolbox for modelling, simulation and verification of real-time systems. In (Behrmann et al., 2006) it is stated that the tool is developed to work on "the systems that can be modelled as networks of timed automata extended with integers variables, structured data types, user defined functions, and channel synchronization". Here, we will mention only a few features of Uppaal that are most relevant to our discussion. An extensive tutorial on the features of Uppaal can be found in (Behrmann et al., 2006).

We first introduce the timed automaton formalism adopted by Uppaal, which is slightly different than but compatible to our definition of timed automata. The following definition of timed automaton is taken verbatim from (Behrmann et al., 2006).

Definition 7.2.6. A timed automaton is a tuple (L, l_0, C, A, E, I) , where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ -action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \rightarrow B(C)$ assigns invariants to locations. $\square$

The fundamental difference between the two definitions of timed automata in Definition 7.2.2 and in Definition 7.2.6 is that in the latter definition, the one Uppaal adopts, edges are labelled with actions. The actions are used for interaction while composing timed automata. In a network of timed automata which is obtained by composing a number of timed automata, composed automata can synchronize on their common actions, similar to channel synchronizations in CSP (Hoare, 1978). The internal τ -actions are not involved in synchronizations. The semantics of the timed automaton defined in Definition 7.2.6 and the semantics for a network of timed automata can be found in (Behrmann et al., 2006).

The important thing is that a timed automaton defined according to our definition (Definition 7.2.2) can be easily converted to a timed automaton compatible with Uppaal's (Definition 7.2.6) by embedding each edge with the internal τ -action.

Uppaal features several syntactic extensions to the simple timed automata formalism given in Definition 7.2.6, such as constants, bounded integer variables, binary and broadcast channels, urgent synchronizations and locations, arrays, custom types, user functions, etc. Here, we include a simple example taken from (Behrmann et al., 2006) to demonstrate modelling features of Uppaal. In Figure 7.2, a Uppaal system modelling a simple lamp is shown. In Listing 7.1 the declarations for the system are shown. The system is composed of two timed automata, a lamp model and a user model. The timed automata for lamp (Figure 7.2a) has three locations: `off`, `low` and `bright`. A button press by user is expressed by the synchronization channel `press`. The lamp is turned on when the user presses the button and it is turned off by another press. However, when the lamp is off, if the user presses the button rapidly twice then the

lamp is first turned on and then becomes bright. As seen in Figure 7.2, in Uppaal, location names are in maroon (e.g. *bright*), location invariants are in purple (e.g. $y < 10$), channel synchronizations are in light blue (e.g. *press?*), transition guards are in green (e.g. $y \geq 5$) and transition updates are in blue (e.g. $y = 0$). The initial locations are denoted by double circles.

Figure 7.2: A simple lamp modelled in Uppaal (Behrmann et al., 2006).

```

1  chan press;
2  clock y;

```

Listing 7.1: Declarations in Uppaal for the lamp model shown in Figure 7.2.

In Uppaal, for a given network of timed automata, the complete state of the system is defined by the combination of locations of all automata, clock values and the values of the discrete variables. Being able to define integer variables shared by automata in a network of automata enables us to define open timed automata according to Definition 7.2.3. The thing to keep in mind that the value of a shared variable v that is defined as an output for an automaton should only be modified by that automaton. Other automata that declare v as input can only use v in edge guards and for updating their discrete state variables.

In Uppaal there are two ways of analysing a system's behaviour: by simulation or by formal verification. For a system modelled in Uppaal as a network of timed automata, one can use Uppaal to simulate system's behaviour by either choosing transitions manually or letting Uppaal choose them randomly. During the simulation, at a state of the system, one can see the locations where the automata are at, the values of variables and clocks, the synchronizations occurred and the available transitions from that state.

Regarding formal verification, one can use Uppaal's model-checking engine to verify a given system with respect to a desired property. In Uppaal, properties are expressed using Uppaal's query language. As the query language Uppaal uses a simplified version of TCTL (Timed Computation Tree Logic, (Alur et al., 1990)). A query is composed

of a path and a state formula. A state formula describes the individual states. It is a predicate which can be evaluated on a single state of the system without considering the dynamic behaviour of the system. For instance, $y < 10$ and $\text{Lamp}.\text{low}$ are valid state formulas for the system depicted in Figure 7.2. A path formula is used to quantify a state formula over paths and traces of a model. A path formula is composed of a path quantifier and a path-specific quantifier. Path quantifiers are as follows:

- $\mathbb{A}\varphi$: *All*. φ is satisfied in all the paths.
- $\mathbb{E}\varphi$: *Exists*. φ is satisfied in at least a path.

In a path formula, after a path quantifier, a path-specific quantifier comes. Path-specific quantifiers are as follows:

- $\Box\varphi$: *Always*. φ is always satisfied along the path.
- $\Diamond\varphi$: *Eventually*. φ is eventually satisfied along the path.
- $\oslash \rightarrow \varphi$: *Leads-to*. Along the path, if $\oslash$ is satisfied at a state then φ is satisfied eventually.

We stated that using Uppaal, we can express open timed automata. However, a A/G contract $C = (A, G)$, a switching contract in this case, is composed of at least two assertions, i.e. the assumption A and the guarantee G assertions. Furthermore, A and G can themselves be logical combinations of individual assertions. Therefore, it is essential to be able to perform simulations and verifications in Uppaal on the combination assertions rather than on a single assertion in isolation. As we stated, in Uppaal it is possible to define a network of timed automata by composing several individual timed automata. This enables us to simulate and verify several assertions together.

First, we need to explain the requirements to compose two open timed automata. Recall that, we required that two open hybrid automata to be compatible in order to compose them (See Definition 5.2.7). Since an open timed automaton is also an open hybrid automaton, this compatibility requirement also applies to open timed automata:

Definition 7.2.7. Two open timed automata OTA_1 and OTA_2 are *compatible* if the following three conditions hold:

1. $(Q_1 \cup X_1) \cap (Q_2 \cup X_2 \cup V_2 \cup Y_2) = \emptyset$
2. $(Q_2 \cup X_2) \cap (Q_1 \cup X_1 \cup V_1 \cup Y_1) = \emptyset$
3. $Y_1 \cap Y_2 = \emptyset$

□

The first and second items in Definition 7.2.7 state that the internal variables $Q \cup X$ of one automaton should be disjoint from all the variables $Q \cup X \cup V \cup Y$ of the other. This is required since internal variables of an automaton are hidden from outside. The third item states that the output variables of the two automata should be disjoint. This is required since an output variable cannot be controlled by both automata.

Now, we need to explain what does composing two open timed automata correspond to in terms of the sets of behaviours that the individual automata specifies. Let OTA_1 and OTA_2 be two compatible timed automata. The input and output variables of OTA_1

and OTA_2 are V_1 , V_2 , Y_1 , and Y_2 such that $Y_1 \cap Y_2 = \emptyset$. Let $OTA_{1||2}$ be the composition of OTA_1 and OTA_2 , i.e. $OTA_{1||2} = OTA_1 || OTA_2$. The input and output variables of $OTA_{1||2}$ are $V_{1||2} = (V_1 \cup V_2) \setminus (Y_1 \cup Y_2)$ and $Y_{1||2} = Y_1 \cup Y_2$. Note that a trace t is included in $OTA_{1||2}$ if and only if it is a trace of OTA_1 and OTA_2 when projected onto their alphabets:

$$t \in \text{Trace}(OTA_{1||2}) \iff pr_{V_1 \cup Y_1}(t) \in \text{Trace}(OTA_1) \wedge pr_{V_2 \cup Y_2}(t) \in \text{Trace}(OTA_2) \quad (7.2)$$

We can conclude directly from Equation (7.2) that the intersection of sets that are inverse projections of traces of OTA_1 and OTA_2 gives the traces of the composition automaton $OTA_{1||2}$:

$$\text{Trace}(OTA_{1||2}) = pr_{V_{1||2} \cup Y_{1||2}}^{-1}(\text{Trace}(OTA_1)) \cap pr_{V_{1||2} \cup Y_{1||2}}^{-1}(\text{Trace}(OTA_2)) \quad (7.3)$$

The equality in Equation (7.3) implies that for an alphabet $P = V_{1||2} \cup Y_{1||2}$, the assertion defined by the composition of two open timed automata equals to the conjunction of the individual assertions of the constituent automata projected to P :

$$\text{Asrt}_P(OTA_{1||2}) = \text{Asrt}_P(OTA_1) \wedge \text{Asrt}_P(OTA_2) \quad (7.4)$$

The above result means that one can compose two assertions expressed by open timed automata in Uppaal and the resulting composed timed automaton expresses the conjunction of the two assertions.

Note that the result in Equation (7.4) is valid only if the constituent automata do not share any output variables. However, this is still useful when the assertions can be formulated such that each output is constrained by only one assertion. In order to demonstrate, let $C = (A_1 \wedge A_2, G_1 \wedge G_2)$ be a contract with alphabet $P = P_{in} \cup P_{out}$. Let also that all the assertions, A_1 , A_2 , G_1 and G_2 , are formulated as open timed automata. We already know that $Y_{A_1} \cap P_{out} = Y_{A_2} \cap P_{out} = \emptyset$ and $Y_{G_1} \cap P_{in} = Y_{G_2} \cap P_{in} = \emptyset$ since assumptions/guarantees should not constrain output/input variables. Let's also assume that each variable in P is bound by at least one assertion. Furthermore, in order to employ Uppaal for simulation and verification, we need that $Y_{A_1} \cap Y_{A_2} = \emptyset$ and $Y_{G_1} \cap Y_{G_2} = \emptyset$, each variable in P is bound by at most one assertion. Then, in Uppaal we can compose these timed automata to form a network of timed automata. On this system we can perform simulations and verification of desired properties in order to see if the formulated contract is valid or not (See Section 7.4 for validation of switching contract).

The above scenario is ideal in the sense that all parts, that is all the assertions included, of the contract is simulated and verified together. However, one may need to validate some parts of the contract C , such as assumptions, $A_1 \wedge A_2$, or guarantees, $G_1 \wedge G_2$, in isolation, or for some input ports there might be no assumptions. In such a case, some variables in P will be unbound in Uppaal. During simulation and verification, values of the unbound variables are not defined or stay at whatever they initially are. However, these variables are still needed since they might be occurring in the invariants

and edge guards of the constituent automata. What we need to have in simulation and verification for unbound variables to be non-deterministically assigned/evaluated.

Since all the variables in P are discrete valued, we can achieve this using supplementary automata to bind the unbound variables in P . For the contract C given above, let's assume that we want to simulate A_1 in isolation. Then for each variable $v \in P$ that is unbound by A_1 , that is $v \notin Y_{A_1}$ we need to have a supplementary automata.

```

1 typedef int [10,15] v_type; // type of v
2 v_type v; // the variable v
3 const int d = 6; // number of values in the domain of v
4 v_type v_values[d] = {10,11,12,13,14,15}; // array that keeps the values in the domain of v

```

Listing 7.2: Declarations in Uppaal to be used by the accompanying automata in Figure 7.3.

Figure 7.3: A timed automata template in Uppaal modelling accompanying automata for variable v declared in Listing 7.2.

The declarations and the accompanying automata for such a variable is shown in Listing 7.2 and in Figure 7.3. In this example we have the domain of v is $D(v) = \{10, 11, 12, 13, 14, 15\}$. In Listing 7.2, we defined the domain of v by a new type v_type . Then, using v_type , we declared the variable v itself. In the last two lines, we declared the array v_values that keeps all the possible values in the domain of v .

In Figure 7.3, the accompanying automata for binding v is shown. On the only edge we non-deterministically select and assign a value to v . Note that this edge can be taken any time.

7.3 Case: Mode Switching of the Robot

In this section, we demonstrate how we construct the mode-switching contract from a given set of mode switching requirements. For the switching component of the robot case, the initial requirements are shown in Table 7.1.

For the demonstration purposes we use a switching component that employs only the m_1 and m_2 modes shown in Figure 6.3. Recall that the functional description of these modes are

m_1 : The stop mode. Robot stays still.

m_2 : The line-follow mode. Robot drives following the line.

Figure 7.4: Ports of the mode-switching component.

This scaled down switching component treats the cases where the robot faces a junction or it is blocked by an obstacle as unrecoverable interruptions to the robot's behavior. This is made clear by the requirements listed in Table 7.1.

- R_1 . On the u_rep port switching component reports the current status of the robot which can be *NOTSET*, *STOPPED*, *DRIVING* or *INTERRUPTED*. Initially u_rep is set to *NOTSET*.
- R_2 . On the u_com port, switching component reads the user commands which can be *DRIVE*, *STOP* or *NOTSET*. Initially u_com is set to *NOTSET*.
- R_3 . Only when u_rep is *STOPPED*, u_com can be set to *DRIVE*. Only when u_rep is *DRIVING*, u_com can be set to *STOP*.
- R_4 . On the $init$ port the system is initialized or stopped. The values of the $init$ port can be *IDLE* or *START*. Initially $init$ is set to *IDLE*.
- R_5 . The $ev_rep_{m_2}$ port indicates the status of the m_2 mode when it is active. Initially it is set to *N*. When m_2 mode is active $ev_rep_{m_2}$ can be set to *B* or *J*. When m_2 mode is deactivated, $ev_rep_{m_2}$ is set to *N* within t_0 time units.
- R_6 . System is initiated if the $init$ port is set to *START*. When the value of $init$ goes from *IDLE* to *START*, the M_1 mode is activated and u_rep is set to *STOPPED* within t_1 time.
- R_7 . While u_rep is set to *STOPPED*, if u_com is set to *DRIVE*, then m_2 is activated and u_rep is set to *DRIVING* within t_2 time.
- R_8 . While u_rep is set to *DRIVING*, if u_com is set to *STOP* then m_2 mode is deactivated, M_1 mode is activated and u_rep is set to *STOPPED* within t_3 time units.
- R_9 . While m_2 mode is active, if $ev_rep_{m_2}$ is set to *B* or *J* then m_2 mode is deactivated, M_1 mode is activated and u_rep is set to *INTERRUPTED* within t_4 time units.

Table 7.1: Textual requirements of mode-switching behaviour.

Based on the initial requirements the ports of the switching component need to be identified. In Figure 7.4, the mode-switching component, its neighbouring components and the ports between them are shown. The set of ports of the switching component for the robot is defined as follows:

$$P^{sw} = P^{sw}_{os} \cup P^{sw}_{mc}$$

According to Table 7.1, the sets of ports to/from other systems and modes are defined as follows:

- $P^{sw}_{os} \supseteq \{u_com, u_rep, init\}$, the set of ports to/from other systems
- $P^{sw}_{mc} \supseteq \{act_\chi : \chi \in \{1, 2\}\} \cup \{ev_rep_\chi : \chi \in \{1, 2\}\}$, the set of ports to/from modes

The input and output ports of the mode-switching component are as follows:

- $P_{in}^{sw} \supseteq \{u_com, init\} \cup \{ev_rep_\chi : \chi \in \{1, 2\}\}$, the set of input ports
- $P_{out}^{sw} \supseteq \{u_rep\} \cup \{act_\chi : \chi \in \{1, 2\}\}$, the set of output ports

Note that we only define the ports that are relevant to the modes m_1 and m_2 . For that reason, we define here only the subsets for the port sets of the mode-switching component.

According to Table 7.1, the domains for the ports in P^{sw} are defined as follows:

- $D(init) = \{START, IDLE\}$
- $D(act_\chi) = \{ON_\chi, OFF_\chi\}$
- $D(ev_rep_{M2}) = \{N, B, \}$
- $D(u_com) \supseteq \{NOTSET, DRIVE, STOP\}$
- $D(u_rep) \supseteq \{NOTSET, STOPPED, DRIVING, INTERRUPTED\}$

Inspecting the requirement items in Table 7.1, we see that the items R_2 , R_3 , R_4 and R_5 are the assumptions, and the rest of the items are the guarantees of the switching component. The items R_2 , R_3 and R_4 relate to the ports $init$ and u_com , the output ports of other systems in Figure 7.4. These requirements are formalized by the assumption assertion A_1 shown in Figure 7.6. The item R_5 relate to the output port of the mode m_2 , so it is formulated separately by A_2 shown in Figure 7.7. The rest of the requirement items, the items R_1 , R_6 , R_7 , R_8 and R_9 , are formulated by a single guarantee assertion G_1 shown in Figure 7.8. Note that all the assertions are expressed by timed automata models in Uppaal. In the Uppaal models of the assertions we use an urgent channel to enforce some of the edges to be urgent as well. The urgent channel is defined as follows:

```
1 urgent chan urgChan;
```

Listing 7.3: Declarations in Uppaal to be used by the accompanying automata in Figure 8.8.

Figure 7.5: The supplementary process for having urgent edges in Uppaal.

We also need a supplementary process shown in Figure 7.5 that accepts a transition with `urgChan` any time. Then the mode-switching contract can be defined as an A/G contract $C_{sw} = (A_1 \wedge A_2, G_1)$.

The obligations or behaviours of C_{sw} are defined by:

$$B_{C_{sw}} = \text{Asrt}(OTA_{A_1} \parallel OTA_{A_2} \parallel OTA_{G_1})$$

where OTA_{A_1} , OTA_{A_2} and OTA_{G_1} denote the open timed automata shown in Figure 7.6, Figure 7.7 and Figure 7.8.

Figure 7.6: The Uppaal timed automaton for A_1 assertion of the switching contract.

Figure 7.7: The Uppaal timed automaton for A_2 assertion of the switching contract.

Figure 7.8: The Uppaal timed automaton for G_1 assertion of the switching contract.

7.4 Validity of Mode-Switching Contract

In this section, we state and demonstrate how we can assess the validity of a mode-switching contract. In Section 5.5, we stated that we have three validity criteria for contract-based system specification: completeness, correctness and consistency. We used these criteria in Subsection 6.1.4 and in Subsection 6.2.3 to assess and improve the validity of mode specification. The same set of criteria apply to mode switching specification as well. In this section, for each criterion:

- We state informal/formal definitions of the validity criterion for a mode-switching contract.
- For each criterion, we provide methods and important points for assessing and improving the criterion on a switching contract.
- We demonstrate these methods and points on the robot case.

7.4.1 Correctness

By correctness of switching contract, we mean that *the switching contract refines a list of requirements given for the mode-switching component* (See Subsection 4.3.2). That is, the switching contract C_{sw} for a switching component with requirements R_{sw} is correct if and only if:

$$C_{sw} \leq R_{sw}$$

For the A/G contract formalism defined in Chapter 5, this refinement relation is satisfied if and only if obligations of C_{sw} is a subset of the obligations defined by the switching requirements (See Subsection 5.5.1):

$$C_{sw} \leq R_{sw} \iff B_{C_{sw}} \subseteq B_{R_{sw}} \quad (7.5)$$

where

- $B_{C_{sw}} = A_{sw} \cap G_{sw}$ denotes behaviours defined by the switching contract C_{sw} and
- $B_{R_{sw}} = \bigcap_{R_i \in R_{sw}} B_{R_i}$ denotes the behaviours that are defined collectively by the requirements in R_{sw} .

Given that the switching contract C_{sw} is defined in Uppaal, Uppaal's simulation and verification capabilities can be employed to check the right hand side relation in Equation (7.5). However, note that R_{sw} provides only an informal definition of the requirements. As a result the relation at the right hand side of Equation (7.5) cannot be directly checked formally. If one can obtain formal versions of individual requirements that can be expressed in Uppaal's query language mentioned in Subsection 7.2.2, then for those requirements formal verification can be employed.

For our test case, we can formally check correctness of the switching contract C_{sw} by checking if B_{sw} conforms to each of the requirements in Table 7.1:

$$\forall R_i \in R_{sw} : B_{C_{sw}} \subseteq B_{R_i}$$

For demonstration, in order to check if B_{sw} conforms to R_1 , i.e. $B_{sw} \subseteq B_{R_1}$, we can check the Uppaal model for $OTA_{A_1} \parallel OTA_{A_2} \parallel OTA_{G_1}$ if:

1. u_com is only set to the values that correspond to *NOTSET*, *STOPPED*, *DRIVING* or *INTERRUPTED*, and
2. u_com is initially set to the value that correspond to *NOTSET*.

The first item above can be checked by verifying the following TCTL property in Uppaal:

- $A[] ((u_com == NOTSET) \vee (u_com == STOPPED) \vee (u_com == DRIVING) \vee (u_com == INTERRUPTED))$

The second item can be ensured by assigning u_com with *NOTSET* in the global declarations as follows:

```
1 int [0,3] u_rep = NOTSET;
```

Listing 7.4: Initial declaration of u_com in Uppaal.

We omit the explanations regarding the assessment of the correctness for the rest of requirements.

7.4.2 Completeness

By completeness of a switching contract, we mean that *the switching contract captures all the necessary switching behavior expected from the system*. We denote the completeness of switching contract by $\kappa(C_{sw})$. Completeness of switching contract implies the completeness of switching requirements (See Figure 4.8) given that the switching contract is correct:

$$(C_{sw} \leq R_{sw}) \Rightarrow (\kappa(C_{sw}) \Rightarrow \kappa(R_{sw}))$$

As stated in Subsection 4.3.2 and in Section 5.5, there is no formal definition of completeness since there is no formal reference against which it can be checked. However, completeness of the switching specification can be effectively assessed on the switching contract due to the formal semantics and executable nature of the switching contract. If the switching contract is expressed as a network of Uppaal models, just like we have for our robot case, formal verification and simulations on the Uppaal models can be employed to investigate completeness of the switching contract.

A criterion to help assessing completeness is coverage of the domains of the ports of the switching contract. All the ports of the switching contract are discrete valued. Some of the ports have domains of ordered values to express a quantity such as discretized real world value. The rest of the ports would be unordered enumerations for not expressing a quantity but for synchronization across components. For our robot case, all the identified ports are of the latter type. For such ports of synchronization nature, switching contract's coverage of the port domain is a necessity for completeness. Informally, a contract covers the domain of a port if every value in the domain is taken by the port in some of the trajectories included in the obligations of the contract. Formally, for a port p , p 's domain is covered by a switching contract C_{sw} if:

$$\forall v \in D(p), \exists (\tau, w) \in B_{C_{sw}}, v \in RANGE(w)$$

where $RANGE(w)$ denotes the range of function w .

If a switching contract does not cover the domains of the ports that are of synchronization nature, this can be due to two things: either the domain is misidentified or the contract is incomplete.

The coverage of a domain of a port can be easily checked in Uppaal using formal verification. This can be done by formulating and checking Uppaal queries for all values for all ports. For instance, in our robot case, for the port *init*, we can check if it ever takes every value in the port domain by checking the following queries in Uppaal:

- $E \langle \rangle \text{ init} == \text{START}$
- $E \langle \rangle \text{ init} == \text{IDLE}$

7.4.3 Consistency

According to Subsection 5.5.3, the criteria for consistency of a given saturated A/G contract $C = (A, G)$ is as follows:

- C defines consistent implementations and legal environments, i.e. $B = A \cap G \neq \emptyset$

If the switching contract is expressed in Uppaal as a network of Uppaal models, formal verification can be used to check the consistency by directly checking the item above.

Recall that, for our robot case, we formulated the switching contract C_{sw} using the assertions A_1 , A_2 and G_1 , i.e. $C_{sw} = (A_1 \wedge A_2, G_1)$. We formulated the assertions A_1 , A_2 and G_1 as open timed automata models in Uppaal. Then the obligations of C_{sw} is given by the behaviours of the composition of the three open timed automata:

$$B_{C_{sw}} = \text{Asrt}(OTA_{A_1} \parallel OTA_{A_2} \parallel OTA_{G_1})$$

For C_{sw} to be consistent, $B_{C_{sw}}$ should be non-empty. This can be easily checked in Uppaal on $OTA_{A_1} \parallel OTA_{A_2} \parallel OTA_{G_1}$. However, we also require that an assertion is composed of only infinite trajectories (See Section 5.1). As opposed to the assertions defined by observers, assertions defined by a timed automata might have maximal, finite trajectories. Therefore, we also need to ensure that the system $OTA_{A_1} \parallel OTA_{A_2} \parallel OTA_{G_1}$ does not contain any maximal, finite trajectories. To do this, one can formally verify that the system does not contain any deadlocks by running the following query in Uppaal:

- $A[]$ not deadlock

For our robot case, we verified that $OTA_{A_1} \parallel OTA_{A_2} \parallel OTA_{G_1}$ does not have any deadlocks.

Note that, deadlock in Uppaal is defined to be the lack of discrete transitions from a state (Behrmann et al., 2006). In Uppaal, a state for which a discrete transition is not possible but continuous-time progress is possible is still identified as a deadlock. According to our definition of assertions in Section 5.1, an assertion can include infinite trajectories which do not have any discrete transitions after a point in time. However, such a trajectory would cause a deadlock in Uppaal. To eliminate the false negatives caused by such trajectories, one can add self-going, clock-guarded transitions to the final locations of the automata where continual time progress without any discrete

transitions is allowed. For instance in the automaton for G_1 shown in (Figure 7.8, we have such a transition on the left one of the top two locations.

7.5 Conclusions

In this chapter we showed how to specify mode switching behaviour of a system with a formal A/G contract. In order to express switching behaviour, using timed trajectories, a restricted class of hybrid trajectories, is sufficient. In this direction, we presented definitions for timed automata and timed observers to express assertions composed of timed trajectories. Due to the decidability of properties of timed automata and the availability of an efficient model checking tool, Uppaal, using formal verification is possible while assessing validity of a switching contract.

We observed that a switching contract is more complex than a mode contract since it realizes the overall system goals by communicating with other systems and orchestrating the modes (Figure 7.4). Hence, it is both error prone and harder to ensure completeness trying to formulate the switching contract with many observers, as in one per a requirement item, or with a single large observer. We proposed using non-observer assertions which can be composed in Uppaal with certain restrictions on the ports restricted by individual assertions. The composition is a network of timed automata that can be directly simulated and formally checked for properties in Uppaal. We demonstrated construction of switching contract with non-observer assertions on the robot case.

Lastly, we defined validity criteria and stated the points to consider while assessing these criteria. We demonstrated on the case that simulation and formal verification in Uppaal can be employed to check validity of a switching contract.

8

Composition of System Specification Contract

In Chapter 4, we proposed adopting a mode-based architecture for structuring the specification of a control system. According to this architecture, as seen in Figure 4.6 and Figure 4.7, we decompose a system into a mode switching logic and a number of modes, each assigned with a control objective. In Chapters 6 and 7, we presented definitions and methods for specifying mode-switching and mode-composition components as A/G contracts. Due to the differences in nature of behaviours for switching and mode components, we employed different formalisms, methods and tools to construct and validate the mode-switching and mode-composition contracts.

In this chapter, we present the methods to construct and validate the overall system contract. In Section 8.1, we state the properties and formulation of a generic system contract. In Section 8.2, we explain the method to execute together a switching contract composed of non-observer assertions in Uppaal and a mode-composition contract composed of observer assertions in 20-sim. In Section 8.3, we demonstrate the construction of overall system contract for the robot case. Finally, in Section 8.4, we state and explain the validity criteria for the overall system contract and describe the methods to check the validity. In the case of invalidity of the system contract, we propose methods of including additional assumptions to make it valid.

8.1 Generic Overall System Contract

In this section we state the ports and contract formulation of the generic overall system component. A system component is the composition of a switching component and a mode-composition component. As seen in Figure 8.1, ports of a system component is defined by the union:

$$P_{sys} = P_{sw}^{os} \cup P_{mc}^e$$

where P_{sw}^{os} are the set of ports between the switching component and the other systems, and P_{mc}^e are the set of ports between the mode-composition component and the environment.

The system contract $C_{sys} = (A_{sys}, G_{sys})$ is an A/G contract that is the composition of the switching contract and the mode-composition contract:

$$C_{sys} = C_{sw} \otimes C_{mc}$$

Figure 8.1: Ports of the generic system component

Given that both C_{sw} and C_{mc} are saturated contracts, the assumption and the guarantee assertions of C_{sys} are defined as follows:

$$A_{sys} = (A_{sw} \wedge A_{mc}) \vee \neg(G_{sw} \wedge G_{mc})$$

$$G_{sys} = G_{sw} \wedge G_{mc}$$

As stated in Chapters 6 and 7, behaviours included in mode composition and switching contracts are hybrid trajectories and timed trajectories, respectively. In Chapter 7, we also stated that timed trajectory is a restricted form of hybrid trajectory. Consequently, being the composition of mode composition and switching contracts, the overall system contract is composed of behaviours that can be classified as hybrid trajectories (Definition 5.1.4).

8.2 Using Uppaal-Tron for Timed Observers

In Chapter 5, we used hybrid observers to express assertions for the mode-composition contract and performed analysis by simulation in 20-sim. In Chapter 7, we formulated non-observer assertions for the switching contract using open timed automata and used simulation and verification features of Uppaal to analyse the assertions. The overall system contract is the composition of mode composition and switching contracts. So, it can be composed of both hybrid observer-based assertions and non-observer timed-automata assertions. In order to effectively assess the validity of the system contract we need to execute both sort of assertions together.

In this direction, one solution is to translate the assertions from open timed automata to timed observers and perform the analysis in 20-sim. However, this translation is laborious and error prone. Fortunately, due to the computation of state space of timed automata being decidable, we can use an open timed automaton as an observer with the help of an model-based testing algorithm using Uppaal (Hessel et al., 2008). First, we explain the features of a testing extension for Uppaal, namely the Tron tool (Hessel et al., 2008; Larsen et al., 2009), that enables this method. Then, we explain how we use Tron for this purpose.

8.2.1 Tron

We first explain the idea of model-based testing and then provide the details of the tool Tron, which is a testing extension of Uppaal. Model-based testing can be defined

as "the automation of the design of black-box tests" (Utting and Legeard, 2007). The basic idea is, given models of the desired behaviour of implementation, to test an actual implementation, termed as IUT, with the help of a tool that generates the tests, applies them to the IUT and checks whether the IUT's reactions/outputs conform to the behaviour of the given model of implementation.

Figure 8.2: Modeling of IUT and environment in Uppaal. Bottom part of the figure depicts the real setup which includes the real environment (ENV) and the developed system (IUT). The top part depicts the environment and implementation models in Uppaal. Adapted from (Hessel et al., 2008)

In (Hessel et al., 2008), techniques for model-based testing of real-time systems using Uppaal are explained. Uppaal is especially suitable for testing real-time systems rather than untimed systems since it incorporates timing in generation and execution of tests. In order to use Uppaal for model-based testing, the models of the IUT and its environment should be defined in Uppaal as networks of timed automata defined in Definition 7.2.6. The interface between the IUT and its environment should be defined in terms of synchronization actions. This is depicted in Figure 8.2. In the bottom part of the figure, the real setup is shown which includes the real environment (ENV) and the developed system (IUT) and their interaction (input and output). The top part of the figure shows the models in Uppaal. The wavy arrows denote that the E and S correspond to the timed automata models of the environment and the system under test. The arrows tagged with *in* and *out* correspond to the abstraction of the interactions between system and its environment by the synchronization actions in Uppaal.

Model-based testing with Uppaal relies on a particular conformance relation, called *relativized timed conformance*, between two timed automata (Hessel et al., 2008). In (Hessel et al., 2008) relativized conformance relation is explained as follows: "this relation ensures that the implementation shows only the behaviour allowed by the specification. In particular, 1) it is not allowed to produce an output at a time when such an output is not allowed by the specification, 2) it is not allowed to omit producing an output when such an output is required by the specification." In the testing setup shown in Figure 8.3, this conformance relation is used to generate a test based on the

Figure 8.3: Typical test setup with Uppaal and Tron. The Uppaal-Tron tester emulates the environment based on E and feeds the IUT with inputs. The outputs generated by implementation is monitored by the tester to check if it conforms to the implementation model S . Adapted from (Hessel et al., 2008)

environment model, and to decide if the output generated by the implementation under test conforms to the implementation specification or not.

8.2.2 Timed Observers in Uppaal

The online test setup shown in Figure 8.3 performs two different functions: environment emulation and implementation monitoring. Environment emulation is the function of replacing the real environment of IUT by generating inputs and feeding them to IUT. Implementation monitoring is the function of passively observing the interaction between the IUT and its real/emulated environment, and determining whether the observed behaviour conforms to the specification.

Using Uppaal-Tron environment emulation and implementation monitoring can be performed together or separately. In order to be able to use specification expressed as timed automata in Uppaal as timed observers, we use the function of implementation monitoring of Uppaal-Tron. Use of Uppaal-Tron for pure implementation monitoring is depicted in Figure 8.4.

Comparing Figure 8.4 with Figure 8.2, we see that the environment model E is replaced by O . The process O denotes the least discriminating process that silently consumes all the input and output actions. The use of a totally passive environment model is due to that the inputs to IUT is generated by the real environment. Note that, for the same reason, the inputs and outputs are both provided by the real environment-IUT setup and consumed by the monitor.

Now we explain how to employ Uppaal-Tron's monitoring in order to obtain a timed observer for an assertion expressed with an open timed automaton. Let L be an assertion for a component M with alphabet $P = P_{in} \cup P_{out}$. Let L be expressed by an open timed automaton OTA_L where input and output variables of OTA_L are such that $V_{OTA_L} \subseteq P_{in}$ and $Y_{OTA_L} \subseteq P_{out}$. Then the set of behaviors contained in L is defined by the traces of OTA_L : $L = \text{Asr}_{TP}(OTA_L)$ (See Subsection 7.2.1).

Figure 8.4: Use of Uppaal-Tron only for implementation monitoring. IUT interacts with the real environment. The inputs and outputs of IUT are monitored by Uppaal-Tron to check if IUT conforms to the implementation model S . Adapted from (Hessel et al., 2008)

OTA_L only binds its input variables, the variables included in V_{OTA_L} . We need to bind all the remaining variables in P , as described in Subsection 7.2.2, in order to be able to make Uppaal generate the state space of OTA_L , which is required while using Uppaal-Tron as a monitor. Let OTA'_L be the Uppaal system resulting from combining OTA_L with the supplementary automata necessary to bind all the variables included in P/V_{OTA_L} .

Note that our open timed automata definition Definition 7.2.3, according to which OTA_L is formulated, does not have input/output synchronization actions on the edges. Interaction is instead based on input and output variables shared among the composed automata. On the contrary, Uppaal-Tron relies on input/output actions with the possibility of variable updates being embedded in the actions. Thus, it is not possible to use OTA'_L in the place of the implementation specification S in Figure 8.4.

To devise a workaround to this problem, we first need to understand what sort of behaviours an observer will be observing when it used within a hybrid system simulation tool. Recall that our focus is on timed trajectories which are traces of an open timed automaton defined according to Definition 7.2.3. A timed trajectory is a restricted form of hybrid trajectory defined in Definition 5.1.4. In a timed trajectory (τ, w) defined over a collection of variables W , all the variables are discrete valued and values of all the variables are constant for all the intervals in the hybrid time set $\tau = \{I_i\}_0^N$. In a hybrid system simulation tool, such as 20-sim, where we would like to use timed observers, we assume that, the behaviours will be timed trajectories where all the intervals in the hybrid time set are non-zero:

Definition 8.2.1. A *timed trajectory* (τ, w) with *non-zero intervals* is a hybrid trajectory consisting of a hybrid time set $\tau = \{I_i\}_0^N$ and a sequence of functions $w = \{w_i\}_0^N$ with $w_i : I_i \rightarrow \mathbf{W}$ such that:

1. Every $w_i \in w$ is a constant function.
2. Every $I_i = [\tau_i, \tau'_i] \in \tau$ is a non-zero interval, i.e. $\tau'_i - \tau_i > 0$.

□

Then, the aim is to employ Tron in using an open timed automaton, in particular the automaton OTA'_L from above, as a timed observer that will observe the behaviours according to Definition 8.2.1. To this end, we compose OTA'_L with a supplementary timed automaton R in Uppaal. The monitoring setup that uses the supplement process R is depicted in Figure 8.5. R offers a synchronization action called set_vars for setting the values of all the variables in P at any time. And R enforces that between two consecutive setting actions, the variables in P have their values as set by the earlier set action.

Compare the depiction of monitor in Figure 8.5 with the one in Figure 8.4. In Figure 8.5 we explicitly draw the input and output variables included in P . The dashed arrows denote the reading and writing to variables whereas the solid arrows denote the synchronization actions. The passive process O for this setup is shown in Figure 8.6.

Figure 8.5: Uppaal monitor setup for open timed automata with input/output variables.

set_vars?

Figure 8.6: The passive process O .

Figure 8.7: Example OTA_L

In order to demonstrate the definition and use of the supplementary process R , we use a simple assertion modelled in Uppaal. Let L be an assertion with $P_L^{in} = \{\}$ and

```

1 // type of observer variables
2 typedef int [0,2] var_type;
3
4 // observer variables
5 const int vars_count = 2;
6 var_type v = 2;
7 var_type y = 0;
8
9 // broadcast channel for first set variables
10 urgent broadcast chan initial_sync;
11
12 clock t1;

```

Listing 8.1: Declarations of variables for OTA_L

```

1 // the channel to set observer variables
2 chan set_vars;
3
4 // channel with higher priority
5 urgent chan prior_ch;
6 chan priority default < prior_ch;

```

Listing 8.2: Global declarations for the supplementary process R in Figure 8.8

$P_L^{out} = \{v, y\}$ with domains $D(v) = D(y) = \{0, 1, 2\}$. L is expressed with the Uppaal model OTA_L seen in Figure 8.7. In Listing 8.1, the global declaration and initialization of the variables used in OTA_L are shown. As seen in Figure 8.7, after synchronizing on the `initial_sync` channel, OTA_L assigns v and y with certain values on the transitions to/from s_1 and s_2 locations. (See Section 7.2 for the explanation of having an initial synchronization on `initial_sync`.) On the s_2 location there is also an invariant on clock t_1 .

In Figure 8.8, the supplementary process R to use OTA_L as an observer is shown. The global and local declarations for R are shown in Listing 8.2 and Listing 8.3, respectively. On the example process R in Figure 8.8, we explain how a supplementary process R works.

R has four locations, l_1 , l_2 , l_3 and l_4 . Location l_1 is urgent, on the edge from l_1 to l_2 , an initial `set_vars` action should occur without any delay, storing the initial values of the observer variables in v_s and y_s declared in Listing 8.3. On this edge v_s and y_s can be set to any value of `var_type`. In Uppaal this is done by using the non-deterministic select expressions on formal parameters v_i and y_i of type `var_type`, which are shown in yellow font in Figure 8.8. The thing to note here is that, when we employ Tron, the action `set_vars` is called by the IUT that is observed and the actual values of the variables v_s and y_s are defined by IUT. Then, if it is possible, Tron will bind v_s and y_s to those values in the state space defined by the model in Uppaal.

On the edge from location l_2 to l_3 , the process R synchronizes with the specification

```

1 // helper variables
2 var_type v_s, v_old, y_s, y_old;
3
4 // indexes observed variables are: 0 -> v, 1 -> y
5 const bool vars_none_updated[vars_count] = {false, false};
6 const bool vars_all_updated[vars_count] = {true, true};
7 bool vars_update_flags[vars_count] = vars_none_updated;
8
9 // check a single observer variables against its set counterpart
10 bool check_single_var_set(int[0,vars_count-1] v_index) {
11     if ((v_index == 0 && v == v_s) || (v_index == 1 && y == y_s))
12         return true;
13     else
14         return false;
15 }
16
17 // check all the observer variables against their set counterparts
18 bool check_all_vars_set() {
19     if (v == v_s && y == y_s)
20         return true;
21     else
22         return false;
23 }
24
25 // check all the observer variables against their set and old counterparts
26 bool check_vars_set_old() {
27     if ((v == v_s || (v == v_old && !vars_update_flags[0])) &&
28         (y == y_s || (y == y_old && !vars_update_flags[1])))
29         return true;
30     else
31         return false;
32 }
33
34 // assign all the set variables with the given arguments
35 void assign_vars_set(var_type v_i, var_type y_i) {
36     v_s = v_i;
37     y_s = y_i;
38 }
39
40 // assign all the old variables with the actual variables
41 void assign_vars_old() {
42     v_old = v;
43     y_old = y;
44 }

```

Listing 8.3: Local declarations in Uppaal for the supplementary process R in Figure 8.8

Figure 8.8: The supplementary process R defined for the variable declarations in Listing 7.3

processes on the broadcast channel initial_sync . In location I3, the values of the input and output variables, v and y in our example, must stay equal to the values set at the latest set_vars action. Note that v and y are only set by OTA_L . But the invariant of I3 of R does not allow them to be set to anything but v_s and y_s . Initially, this can be ensured by OTA_L by either having the input and output variables initialized accordingly in global declarations or setting them on the edge with initial_sync channel accordingly.

On the edge from location I3 to I4, all but the first set_vars actions handled. Again, on this edge v_s and y_s can be set to any value in var_type . Also, the values of v and y are stored in v_old and y_old and vars_update_flags array is set all false.

At location I4, which is an urgent location, the invariant allows OTA_L to set v and y either to their current values defined by v_old and y_old or the new values defined by v_s and y_s . Delaying is only possible by going back to the location I3 which is only possible if v and y are set to the new values. Also, once a variable is set to its new value, it cannot go back to its old value. This is ensured by taking the self-going edge on location I4 and setting the flag in vars_update_flags corresponding to the updated variable. Setting update flags by taking this edge as soon as possible, without any interleavings is ensured by synchronizing on the channel prior_ch which has higher priority (See Listing 8.2). The edge from the location I4 to I3 is only taken if all the observer variables are set to new values defined by v_s and y_s .

Now, think of the use of Tron with the monitor set-up in Figure 8.5 for observing an IUT. The IUT can be an actual implementation or it can be simulated. UIT can do two

Figure 8.9: A sample block diagram in 20-sim with a trace source and a timed observer, which is connected to Uppaal-Tron externally.

things: setting values of the variables and delaying for some time. For setting values, IUT sends out an `set_vars` action with a set of values for the variables in P_L . If, for any variable, the values set are out of the domain of the variable this action will not be accepted by Tron, i.e. Tron will report an error, due to the select expression on the edges from l1 to l2 and from 3 to l4 of R . Otherwise the `set_vars` will be accepted by Tron and R will go the location l2 or l4. At this point, if OTA_L is capable of setting all the variables to the specified values, R 's going to the location l3 will be in the state space computed by Uppaal, and a delay or a next `set_vars` action might be possible when R is at location l3. Otherwise, in the next delay or a next `set_vars` action performed by IUT, the state of the monitor will be undefined and Tron will report an error. If through the test or simulation with an observer, Tron does not give an error, then it means that the observer accepts the observed behaviour.

8.2.3 Connecting Tron and 20-sim

Remember that our aim in using Tron+Uppaal is to use an assertion expressed by a network of open timed automata in Uppaal as a timed observer in a hybrid system simulation tool. This is crucial to analyse the assertions expressed by timed automata and the ones expressed by hybrid automata together by simulating them next to each other. In the previous part, we explained how we can use a model in Uppaal as a timed observer with the help of Tron extension. Now, we explain how one can connect 20-sim and Tron to employ such observers in simulations performed in 20-sim.

During a simulation in 20-sim, we use 20-sim's capability of calling external DLL functions in order to connect with Tron. At the 20-sim side, a timed observer is expressed with a submodel. In Figure 8.9, a sample 20-sim block diagram is shown with a timed observer submodel attached to a trace source which can be a submodel of the system to be simulated as in Figure 6.10 on page 97. For a simulation run, the timed observer submodel in Figure 8.9 acts as a gateway to the Uppaal-Tron monitoring construction, which we explained in Subsection 8.2.2, by sending over values of the monitored variables and time delay amounts.

In Listing 8.4, a sample 20-sim implementation for a timed-observer submodel is shown. At every simulation step, an observer model does the following two main things whether it is a delay cycle or there is a change in the values of the observer inputs:

1. Delay for the time passed since the last execution.

<pre> 1 submodel TO 2 type submodel 3 ports 4 signal in input [2,1]; 5 signal out output; 6 end; 7 8 parameters 9 // name of the dll file to connect Tron 10 string dll_name = 'trondll.dll'; 11 12 // the dll delay function 13 string delay_function = 'simDelay'; 14 15 // the dll observer input function 16 string input_function = 'simVarSet'; 17 18 // max delay 19 integer delay_period = 100; 20 21 variables 22 // DELAY VARIABLES 23 // time of the last call 24 real last_call {time, s}; 25 26 // last period 27 real last_interval; 28 29 // result of delay call 30 real delayResult; 31 32 // next delay time 33 integer next_delay_cycle; 34 35 // delay event flag 36 boolean at_delay; 37 38 // SETVARS VARIABLES 39 // difference between current 40 // and previous input 41 real input_diff_norm; 42 43 // input change event flag 44 boolean input_change; 45 46 // result of setting variables 47 real setResult; 48 49 // time zero event flag 50 boolean at_zero; </pre>	<pre> 51 initialequations 52 last_call = 0; 53 last_interval = 0; 54 next_delay_cycle = 0; 55 at_delay = false; 56 57 code 58 // detect any changes in the input 59 input_diff_norm = norm(dly(input,0) - input); 60 input_change = event(if input_diff_norm == 0 61 then -1 else 1 end); 62 at_zero = timeevent(0); 63 at_delay = timeevent(next_delay_cycle * delay_period); 64 65 if (at_delay) then 66 next_delay_cycle = next_delay_cycle + 1; 67 end; 68 69 if major then 70 // at the end of every delay period and at input changes 71 // issue a delay 72 if ((input_change and (input_diff_norm <> 0)) 73 or at_delay) then 74 // compute the time passed since the last execution 75 last_interval = time_var - last_call; 76 77 // make a dll delay call to delay from the last execution 78 delayResult = dll(dll_name, delay_function, last_interval); 79 80 last_call = time; 81 else 82 delayResult = true; 83 end; 84 end; 85 86 if major then 87 // at the beginning and at input changes 88 // make a call to set variables 89 if ((input_change and (input_diff_norm <> 0)) 90 or at_zero) then 91 // make a dll output call to set observer inputs 92 setResult = dll (dll_name, input_function, input); 93 else 94 setResult = true; 95 end; 96 end; 97 98 output = delayResult and setResult; </pre>
---	---

Listing 8.4: The 20-sim implementation of a timed observer.

Figure 8.10: Overview of the connection between Uppaal-Tron and 20-sim.

2. Set the inputs of the observer, i.e. the values of the input and output variables of the assertion that the observer represents.

Delay cycles are series of points in time when we force the observer to make Tron calls even if the observer input values has not been changed. The time between two delay cycles is stated by the fixed delay period defined at line 19. Calling Tron periodically in the case of stable observer inputs ensures conformance of the simulation time and the time delays allowed by the timed observer defined in Uppaal.

There are four parts in the **code** section of Listing 8.4. In the event part, between the lines 59-63, the events are created to force the simulation engine to perform calculations at the beginning of the simulation, at every delay cycle and whenever a change in the values of the observer inputs occurs. In the part between the lines 69-84, the time passed since the last calculation is computed, the last calculation time is updated and a call to the delay function of the external DLL module is made, provided with the amount of time passed since the last calculation as a parameter. In the part between lines 86-96, there is the call to the observer input function of the external DLL module, provided with the values of all the observer input variables as parameters.

Note that our observer implementation assumes that the observer code is executed once at every simulation time step of 20-sim simulation engine, where the exchange between 20-sim and Tron should happen. 20-sim implements various integration algorithms which have exactly one major integration step and zero or more minor integrations steps. For an integration algorithm, the simulation engine produces the result of a time step at the major integration step, possibly using the results of the minor integration steps in case a multi-step algorithm is used. To accommodate for different integration algorithms, indicated by the conditional clauses at lines 69 and 86 in Listing 8.4, we call DLL functions to issue a delay or an update for the variables only at major steps.

An overview of the connection between Uppaal-Tron and 20-sim is depicted in Figure 8.10. The two ends of the figure are the components of Uppaal-Tron and 20-sim. Next to 20-sim, there is the DLL module that implements the external functions to send delay and variable set requests to a test adapter. The test adapter configures the Tron in the beginning of a simulation. During the simulation, the test adapter either issues a delay to Tron with the help of a virtual clock or issues the `set_vars` ac-

tion (see Figure 8.5 and Figure 8.8) to set the input variables of the observer. Delays are issued using a virtual clock in order to accommodate for the simulation time rather than the real time. Here we omit the implementation details of the test adapter and external 20-sim functions.

In Figure 8.11 the plot of an example simulation in 20-sim with an observer model that calls Tron is shown. The observer is based on the Uppaal model shown in Figure 8.7. The top two plots in Figure 8.11 depict the number of delay and variables set calls respectively. A delay call is issued at every 10s and whenever a change in the observer input variables occurs. A variable set call is issued initially and whenever a change in the observer input variables occurs. In the figure, the values for the observer input variables v , in green, and y , in black, are shown. Below them the event for input change is plotted. Note that the simulated trajectories for the observer values conforms to the example Uppaal specification shown in Figure 8.7. In Figure 8.12 and Figure 8.13, outputs of the 20-sim/Tron adapter and Tron processes corresponding to the simulation in Figure 8.11 are shown. Both output windows show no error since the simulation conforms to the associated Uppaal specification.

Whenever, the values sent by 20-sim to Tron do not conform to the associated Uppaal specification, the 20-sim/Tron adapter and Tron execution windows indicate an error and the simulation in 20-sim stops. Such an execution with an error can be seen in Figures 8.14, 8.15 and 8.16 depicting 20-sim simulation, 20-sim/Tron adapter execution and Tron execution, respectively. The error is caused by setting $v = 3$ at 25s in Figure 8.16 which does not conform to the example Uppaal specification shown in Figure 8.7. As it can be seen in Figures 8.15 and 8.16, this error is revealed when, according to the simulation, 20-sim observer issues a delay call at 30s.

8.3 Case: Overall Robot Contract

The contract formulation for the overall robot contract C_{sys} is the composition:

$$C_{sys} = C_{sw} \otimes C_{mc}$$

where

- C_{sw} denotes the switching contract for the robot case defined in Chapter 7
- C_{mc} denotes the mode-composition contract for the robot case defined in Chapter 6

Using the contract algebra presented in Subsection 2.2.2, we can express C_{sys} in terms of the assumption and guarantee assertions of C_{sw} and C_{mc} . However, note that, in order to apply the contract algebra, we need to be able to saturate the contracts, which require handling of complement of assertions.

In Section 8.2, we stated that a switching contract formulated using non-observer assertions in Uppaal can be used as an observer-based contract in 20-sim. This enables us to have a saturated version of switching contract C_{sw} , which is not the case with non-observer assertions formulated in Uppaal.

Figure 8.11: An example simulation in 20-sim with an observer model that calls tron.

```
C:\Users\oguzcan\Documents\workspace\thesis0gz\20sim dll>java -jar C:\Users\oguzcan\Documents\stuff\uppaal\tron\uppaal-tron-1.5-win32\uppaal-tron-1.5-win32\java\dist\tsinadapter.jar -C localhost 8983 -M 0 -T 9993 -S 2300 -O C:\Users\oguzcan\Dropbox\workspace\thesis0gz\observer_config\test\observer_inputs_test.xml
tsinadapter started
Root element: variables
DELAY REQUEST: 0 seconds, 0 microseconds
SET VAR REQUEST: 2 0
IOHandler: Output Variables
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
SET VAR REQUEST: 1 0
IOHandler: Output Variables
DELAY REQUEST: 5 seconds, 0 microseconds
SET VAR REQUEST: 2 0
IOHandler: Output Variables
DELAY REQUEST: 4 seconds, 999999 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 5 seconds, 0 microseconds
SET VAR REQUEST: 0 1
IOHandler: Output Variables
DELAY REQUEST: 4 seconds, 999999 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
```

Figure 8.12: Output of 20-sim/Tron adapter for the simulation shown in Figure 8.11. The adapter output shows the delay and variable set calls made by 20-sim during the simulation.

```

C:\Users\oguzcan\Documents\workspace\thesisOgz\20sim dll>C:\Users\oguzcan\Documents\stuff\uppaal\tron\uppaal-tron-1.5-win32\uppaal-tron-1.5-win32\tron -Q 8983
-F 300000000 -I SocketAdapter -v 9 C:\Users\oguzcan\Documents\stuff\uppaal\tron\uppaal-tron-1.5-win32\uppaal-tron-1.5-win32\java\obs_2vars_w_var_pass_single_action.xml -- localhost 9993
Created clock on port 8983
UPPAAL TRON 1.5 using UPPAAL 4.1.2 (rev. 4351), June 2009
Compiled with i586-mingw32msvc-g++ -Wall -DLIBXML_STATIC -DNDEBUG -O2 -ffloat-store -march=pentiumpro -march=pentium4 -march=prescott -march=pentium-m -DTIGA_MERGE_STATES -DBOOST_DISABLE_THREADS
Copyright (c) 1995 - 2009, Uppsala University and Aalborg University.
All rights reserved.
Options for UPPAAL TRON:
  Search order is breadth first
  Using no space optimisation
  State space representation uses minimal constraint systems
  Observation uncertainties: 0, 0, 0, 0 (microseconds).
  Scheduling latency: 0 microseconds
  Future precomputation: closure(300000000 mtu).
  Input delay extended by: 0
  OS scheduler: non-real-time.
WARNING: partitioning is incomplete, use -i option to investigate.
Emulation invariants: Spec_2, Passive_0, UrgentP, Supp_2, PriorP.
Timeunit: 1000us
Timeout: 100000000mtu
Inputs: (empty)
Outputs: set_vars(v_s,y_s)
TEST in progress i 0%

```

Figure 8.13: Output of Tron engine for the simulation shown in Figure 8.11. Tron output shows Tron related settings and information about the given Uppaal specification.

Figure 8.14: A 20-sim simulation that is stopped at 30s due to call to Tron resulted in an error. The call resulting with error is a delay call made at 30s.

```

C:\Users\oguzcan\Documents\workspace\thesis0gz\20sim_dll>java -jar C:\Users\oguzcan\Documents\stuff\uppaal\tron\uppaal-tron-1.5-win32\uppaal-tron-1.5-win32\java\dist\tsimadapter.jar -C localhost 8983 -M 0 -T 9993 -S 2300 -O C:\Users\oguzcan\Dropbox\workspace\thesis0gz\observer_config\test\observer_inputs_test.xml
tsimadapter started
Root element: variables
DELAY REQUEST: 0 seconds, 0 microseconds
SET VAR REQUEST: 2 0
IOHandler: Output Variables
DELAY REQUEST: 10 seconds, 0 microseconds
DELAY REQUEST: 10 seconds, 0 microseconds
SET VAR REQUEST: 1 0
IOHandler: Output Variables
DELAY REQUEST: 5 seconds, 0 microseconds
SET VAR REQUEST: 3 0
IOHandler: Output Variables
DELAY REQUEST: 4 seconds, 999999 microseconds
TR.Receiver: java.io.EOFException

```

Figure 8.15: Output of 20-sim Tron adapter for the simulation shown in Figure 8.14. The adapter output shows the delay and variable set calls made by 20-sim during the simulation. Comparing with Figure 8.12, execution stops at the first delay call after the erroneous variable set call with values 3 and 0.

Therefore we can express the formulation of $C_{sys} = (A_{sys}, G_{sys})$ with:

$$\begin{aligned}
 A_{sys} &= (A_{sw} \wedge A_{mc}) \vee \neg(G_{sw} \wedge G_{mc}) \\
 G_{sys} &= (G_{sw} \wedge G_{mc})
 \end{aligned}$$

According to this formulation, C_{sys} can be implemented in 20-sim using observers for C_{sw} and C_{mc} as seen in Figure 8.17. The boxes A_{sw} , A_{mc} , G_{sw} and G_{mc} denote the observer models in 20-sim for the assumption and guarantee assertions of C_{sw} and C_{mc} .

8.4 Validity of System Contract

In this section, we state and demonstrate how we can assess the validity of an overall system contract. In Section 5.5, we stated that we have three validity criteria for contract-based system specification: completeness, correctness and consistency. We used these criteria in Chapter 6 and in Chapter 7 to assess and improve the validity of mode composition and mode specification contracts. In order to validate the overall system contract, we take the same set of validity criteria as basis.

In this section, for each criterion:

- We state informal/formal definitions of the validity criterion for an overall system contract.
- For each criterion, we provide methods and important points for assessing and improving the criterion on an overall system contract.
- We demonstrate these methods and points on the robot case.

8.4.1 Correctness

Correctness should be defined with respect to a reference. The initial C_{sys} is correct with respect to $C_{sw} \otimes C_{mc}$ since we defined initial C_{sys} to be equal to this composition.

```

C:\Users\oguzcan\Documents\workspace\thesisOguz\20sim dll>C:\Users\oguzcan\Documents\stuff\uppaal\tron\uppaal-tron-1.5-win32\uppaal-tron-1.5-win32\tron -Q 8983
-F 30000000 -I SocketAdapter -v 9 C:\Users\oguzcan\Documents\stuff\uppaal\tron\uppaal-tron-1.5-win32\uppaal-tron-1.5-win32\java\obs_2vars_w_var_pass_single_acti
on.xml -- localhost 9993
Created clock on port 8983
UPPAAL TRON 1.5 using UPPAAL 4.1.2 (rev. 4351), June 2009
Compiled with i586-mingw32msvc-g++ -Wall -DLIBXML_STATIC -DDEBUG -O2 -ffloat-st
ore -march=pentiumpro -march=pentium4 -march=prescott -march=pentium-m -DIIGA_ME
RGE_STATES -DBOOST_DISABLE_THREADS
Copyright (c) 1995 - 2009, Uppsala University and Aalborg University.
All rights reserved.
Options for UPPAAL TRON:
  Search order is breadth first
  Using no space optimisation
  State space representation uses minimal constraint systems
  Observation uncertainties: 0, 0, 0, 0 (microseconds).
  Scheduling latency: 0 microseconds
  Future precomputation: closure(30000000 mtu).
  Input delay extended by: 0
  OS scheduler: non-real-time.
WARNING: partitioning is incomplete, use -i option to investigate.
Emulation invariants: Spec_2, Passive_0, UrgentP, Supp_2, PriorP.
Timeunit: 1000us
Timeout: 100000000mtu
Inputs: <empty>
Outputs: set_vars(v_s,y_s)
TEST in progress ! 0%Short post-mortem analysis based on last good stateSet(4):
1)
< Spec_2.s1 Passive_0_id24 UrgentP_id20 Supp_2.l4 PriorP_id21 >
t1>=20000, #t>=20000, t1<=20000, t1-#t<=0, #t<=20000, #t-t1<=0 v=2 y=0 v_s=1 v_o
ld=2 y_s=0 y_old=0 Supp_2.vars_update_flags[0]=0 Supp_2.vars_update_flags[1]=0
2)
< Spec_2.s1 Passive_0_id24 UrgentP_id20 Supp_2.l4 PriorP_id21 >
t1>=20000, #t>=20000, t1<=20000, t1-#t<=0, #t<=20000, #t-t1<=0 v=2 y=0 v_s=1 v_o
ld=2 y_s=0 y_old=0 Supp_2.vars_update_flags[0]=0 Supp_2.vars_update_flags[1]=1
3)
< Spec_2.s2 Passive_0_id24 UrgentP_id20 Supp_2.l4 PriorP_id21 >
#t>=20000, t1<=0, t1-#t<=-20000, #t<=20000, #t-t1<=20000 v=1 y=0 v_s=1 v_old=2 y
_s=0 y_old=0 Supp_2.vars_update_flags[0]=0 Supp_2.vars_update_flags[1]=1
4)
< Spec_2.s2 Passive_0_id24 UrgentP_id20 Supp_2.l3 PriorP_id21 >
#t>=20000, t1<=10000, t1-#t<=-20000, #t<=30000, #t-t1<=20000 v=1 y=0 v_s=1 v_old
=2 y_s=0 y_old=0 Supp_2.vars_update_flags[0]=0 Supp_2.vars_update_flags[1]=1

Options for input : <empty>
Options for output : set_vars(v_s=0,y_s=0)[20000;30000], set_vars(v_s=1,y_s=
0)[20000;30000], set_vars(v_s=2,y_s=0)[20000;30000], set_vars(v_s=0,y_s=1)[20
000;30000], set_vars(v_s=1,y_s=1)[20000;30000], set_vars(v_s=2,y_s=1)[20000;30
000], set_vars(v_s=0,y_s=2)[20000;30000], set_vars(v_s=1,y_s=2)[20000;30000],
set_vars(v_s=2,y_s=2)[20000;30000]
Options for internal: prior_ch[20000;20000], -e[20000;20000]
Options for delay : until 30000
Last time-window : [25000;25000]
Got unacceptable output: set_vars(v_s=3,y_s=0)[250000000us at [25000;25000]
Expected outputs were: set_vars(v_s=0,y_s=0)[20000;30000], set_vars(v_s=1,y_s=0
)[20000;30000], set_vars(v_s=2,y_s=0)[20000;30000], set_vars(v_s=0,y_s=1)[200
00;30000], set_vars(v_s=1,y_s=1)[20000;30000], set_vars(v_s=2,y_s=1)[20000;300
00], set_vars(v_s=0,y_s=2)[20000;30000], set_vars(v_s=1,y_s=2)[20000;30000], s
et_vars(v_s=2,y_s=2)[20000;30000]
TEST FAILED: Observed output has wrong variable value(s).
CONN::readchannel: A blocking operation was interrupted by a call to WSACancelBl
ockingCall.

```

Figure 8.16: Output of the Tron engine for the simulation shown in Figure 8.14. Tron output indicates an error during the execution. Details of the state of the given Uppaal specification and input and output options at the point of error can be seen. For detailed explanation of the Tron output refer to (Larsen et al., 2009)

Figure 8.17: The observer model of overall system contract C_{sys} in 20-sim as the composition of observers for C_{sw} and C_{mc}

Correctness of the overall system contract with respect to the overall system requirements R_{sys} is formulated as follows:

$$C_{sys} \preceq R_{sys}$$

This is implied by the correctness of mode-composition contract, switching contract, and the initial requirements decomposition:

$$(\bigwedge (C_{m_i} \preceq R_{m_i}) \wedge (C_{sw} \preceq R_{sw}) \wedge ((\bigcup R_{m_i} \cup R_{sw}) \preceq R_{sys})) \Rightarrow (C_{sys} \preceq R_{sys}) \quad (8.1)$$

since we initially defined $C_{sys} = C_{sw} \otimes C_{mc}$.

8.4.2 Completeness

By completeness of an overall system contract, we mean that *the overall system contract captures all the necessary behaviour expected from the system*. We denote the completeness of system contract C_{sys} by $\kappa(C_{sys})$.

In general, completeness of overall system contract is not implied by the completeness of mode-composition contract and switching contract:

$$(\kappa(C_{mc}) \wedge \kappa(C_{sw})) \not\Rightarrow \kappa(C_{sys})$$

This is due to that C_{sys} might be inconsistent. Another reason is that C_{mc} and C_{sw} might be individually complete with respect to their intended functions initially defined by $R_{m_1} \dots R_{m_n}$ and R_{sw} but their composition might be incomplete due an incompleteness in R_{sys} which cannot be identified by the assessing the modes and switching separately.

Incompleteness of mode-composition contract or of switching contract, implies the incompleteness of the overall system contract:

$$(\neg \kappa(C_{mc}) \vee \neg \kappa(C_{sw})) \Rightarrow \neg \kappa(C_{sys})$$

So first, C_{mc} and C_{sw} need to be complete. Assuming that the mode composition and mode-switching contracts are shown to be consistent earlier, the next step is to show that the overall system contract is consistent as this is a requirement for completeness. After consistency is ensured, that the combination of individual modes and the mode switching logic adequately describes the expected system behaviour should be inspected by simulations performed on the early system models. During simulations, using observer-based formulation of overall system contract C_{sys} enables building system models according to the system specification and testing if a behaviour conforms to the specification or not.

8.4.3 Consistency

According to Subsection 5.5.3 we have the following criterion for consistency of a given saturated A/G contract $C = (A, G)$:

- C defines consistent implementations and legal environments, i.e. $B = A \wedge G \neq \emptyset$

However, the above criterion is too weak for ensuring consistency of overall system contract since it can still hold even if C_{sys} has some behaviour. For instance, the behaviours included in C_{sys} might not involve activation of a certain mode due to some inconsistency but the above criteria would still hold as long as there are behaviours involving some other modes. Therefore, we need stronger consistency criteria that take into account the activation and deactivation of individual modes defined by mode switching logic. We need to ensure that:

1. For any set of modes that can be activated at the same time, the individual modes are consistent with each other and with the mode-switching component.
2. Switchings from a set of modes to another defined by the switching contract are consistent.

In the following parts, we define the above two criteria informally and formally, and describe the methods to assess them on an overall system contract.

Consistency of active modes and mode switching

The sets of modes that can be activated during the operation of a system are defined by the mode-switching contract C_{sw} . For C_{sys} to be consistent, we need to ensure that activation of any set of modes that can be activated by C_{sw} is also possible in C_{sys} and not lost due to inconsistency among mode and mode-switching contracts. Formally we define this criterion as follows:

$$\forall S \in 2^{P_{act}} : \{(act_i, ON) | act_i \in S\} \in RANGE(pr_S(B_{sw})) \implies \{(act_i, ON) | act_i \in S\} \in RANGE(pr_S(B_{sys}))$$

where P_{act} is the set of activation ports of all the modes, $P_{act} = \{act_i | act_i \in P_{sw}\}$.

In order to check the above criterion, we can identify the sets of modes which consist of the modes that can be activated at the same time by C_{sw} . Then, for each such set, we need to inspect by simulation the consistency of C_{sys} in situations where the modes in the set are all active.

The sets of modes that consist of modes that can be activated at the same time in C_{sw} can be obtained with the help of model checking if switching contract C_{sw} is expressed as a Uppaal model. Let $S \in 2^{P_{act}}$ denote a set of modes that we need to check if every mode in S can be activated at the same time by the switching logic. Let Φ the state property for the modes in S being active. Then, we can check the reachability of a state satisfying Φ in the behaviours contained in C_{sw} , i.e. $B_{C_{sw}}$, where each mode in S is active at the same time. The reachability in Uppaal can be checked with the query $\exists \diamond \Phi$.

For our robot case, this can be done by checking the following queries in Uppaal on C_{sw} :

- $E <> (\text{act_m1} == \text{OFF} \text{ and } \text{act_m2} == \text{OFF})$
- $E <> (\text{act_m1} == \text{ON} \text{ and } \text{act_m2} == \text{OFF})$
- $E <> (\text{act_m1} == \text{OFF} \text{ and } \text{act_m2} == \text{ON})$
- $E <> (\text{act_m1} == \text{ON} \text{ and } \text{act_m2} == \text{ON})$

For the robot case, the first three of the above queries are satisfied whereas the last is not since the modes m_1 and m_2 are never active at the same time. In order to find the trajectories in $B_{C_{sw}}$ in which a set of modes are active at the same time, one can simulate $B_{C_{sw}}$ in Uppaal or use witness traces generated by Uppaal. We can use the query $\forall \square \neg \Phi$ and inspect the witness traces generated by Uppaal.

For a set of active modes we can investigate the consistency of modes with each other and with the mode-switching component using simulation. For an overall system contract C_{sys} , let S be the set of activation ports for a set of modes of C_{sys} for which we would like to inspect consistency. Since we are interested in the consistency of C_{sys} when the modes represented by S are active, we can safely ignore the rest of modes. Let C_{sys}^S be the system contract such that

$$C_{sys}^S = C_{sw} \otimes C_{mc}^S$$

where C_{mc}^S is the composition of the modes represented by S :

$$C_{mc}^S = \bigotimes_{\{i | \text{act}_i \in S\}} C_{m_i}$$

Consistency of C_{sys}^S can be formulated using obligations of C_{sys}^S . We need show that

$$\exists tr \in B_{sys}^S : \{(\text{act}_i, \text{ON}) | \text{act}_i \in S\} \in \text{RANGE}(pr_S(tr))$$

The above condition formulates that there is a trajectory in B_{sys}^S in which the modes in S are activated at the same time. For a particular set of modes represented by S , investigating consistency on the above condition can be performed in hybrid system simulations using system models for generating trajectories and the observer for C_{sys}^S . Since the C_{sys}^S is composed of individual observers for assumption and guarantee assertions of modes in S and the mode-switching contract, we can assess the conformance of a given trajectory to individual assertions of mode and mode-switching contracts. Note

Figure 8.18: Conditions to disprove (a) or strengthen (b) consistency with respect to mode-set S .

that, while performing simulations we need to focus on the trajectories where all the modes in S are active.

In general, for a particular set of modes, performing this consistency assessment can be quite complex as individual activation times of the modes defined by switching contract can be different and not fixed. Also, consistency of C_{sys} when a particular set of modes are active might depend on the system state at the point when the set of modes becomes active. Performing the identification of system modes in a way that main system functions are separated, and having the definition of mode and mode-switching contracts in terms of assumptions and guarantees complete can mitigate in this part.

Now, we explain how the consistency of an overall system contract with respect to a set of modes can be assessed. Let L_1 and L_2 be conjunctions of two sets of individual assertions in C_{sys}^S , i.e. $L_1 \supseteq B_{sys}^S$ and $L_2 \supseteq B_{sys}^S$. Then, if for all the trajectories we can show that $L_1 \Rightarrow \neg L_2$ when S holds, this means that

$$\forall tr \in L_1 \wedge L_2 : \{(act_i, ON) | act_i \in S\} \notin RANGE(pr_S(tr))$$

This is illustrated in Figure 8.18a where the time line of such a trace is shown. Showing this for a pair of L_1 and L_2 would show the overall system contract is inconsistent with respect to mode set S .

In Figure 8.18b, a different time line of traces is shown, where at the point of activation of S L_2 holds and when S is active $L_1 \Rightarrow L_2$ holds. For a pair of L_1 and L_2 showing this property does not alone prove consistency of overall system contract with respect to S . However, it would strengthen the confidence on consistency and now L_2 in B_{sys}^S can be discarded while continuing with assessment.

8.4.4 Assessing Validity of Overall Robot Contract

Using the robot case, we demonstrate partial assessment of consistency of overall system contract when a set of modes is active. On our robot case we have two modes, m_1 and m_2 , which cannot be active at the same time according to mode-switching contract. We assess the consistency when the mode m_2 is active.

The set of active modes, which consists of only m_2 , is represented by $S = \{act_{m_2}\}$. The formulation of obligations of C_{sys}^S is

$$B_{sys}^S = B_{sw} \cap B_{m_2}$$

where

$$\begin{aligned} B_{sw} &= A_{sw}^1 \cap A_{sw}^2 \cap G_{sw}^1 \\ B_{m_2} &= A_{m_2}^1 \cap A_{m_2}^2 \cap A_{m_2}^3 \cap A_{m_2}^4 \cap A_{m_2}^5 \cap G_{m_2}^1 \cap G_{m_2}^2 \end{aligned}$$

Note that in the above descriptions we always assume that the alphabets of all the assertions are equalized before applying any set operations. Indeed, the alphabet of the overall system contract for mode set S is equal to the union of alphabets of mode m_2 and mode-switching contracts: $P_{sys}^S = P_{sw} \cup P_{m_2}$.

Ports of the assertions that make up C_{sys}^S are as follows:

$$\begin{aligned} P_{m_2}^{A_1 \cap A_2} &= \{act_{m_2}, v, offline\} \\ P_{m_2}^{A_3} &= \{act_{m_2}, p, \theta, bump, online\} \\ P_{m_2}^{A_4 \cap A_5} &= \{act_{m_2}, p, \theta, w_l, w_r, v, bump\} \\ P_{m_2}^{G_1} &= \{act_{m_2}, online, v\} \\ P_{m_2}^{G_2} &= \{act_{m_2}, rep_{m_2}, bump, junction\} \\ P_{sw}^{A_1} &= \{com_{os}, rep_{os}, init\} \\ P_{sw}^{A_2} &= \{act_{m_2}, rep_{m_2}\} \\ P_{sw}^{G_1} &= \{init, act_{m_1}, act_{m_2}, com_{os}, rep_{os}\} \end{aligned}$$

Observe that, the common ports between C_{sw} and C_{m_2} are act_{m_2} and rep_{m_2} . The port act_{m_2} is an output of switching component and an input of mode m_2 component. The port rep_{m_2} is the exact opposite, an input of switching component and an output of mode m_2 component. Our strategy to show that C_{sys}^S is consistent when mode set represented by S , i.e. mode m_2 , is active is to show that $B_{m_2} \Rightarrow A_{sw}^2$ when S is active. This fits to the scheme depicted in Figure 8.18b when $L_1 = B_{m_2}$ and $L_2 = A_{sw}^2$. In set theoretic terms this will be showing that

$$B_{m_2} \cap L_S \subseteq A_{sw}^2 \cap L_S \quad (8.2)$$

where L_S is the assertion that defines the trajectories where S is active in the end of the trajectory and A_{sw}^2 holds at the point that S become active last time. Using the

observers for $G_{m_2}^2$ and A_{sw}^2 one can partially show in simulations that every trajectory in $G_{m_2}^2$ is also in A_{sw}^2 , that is $G_{m_2}^2 \subseteq A_{sw}^2$ or equivalently $G_{m_2}^2 \Rightarrow A_{sw}^2$. In our case, it is easy to see this implication by manually inspecting both assertions. Since the port rep_{m_2} is only included in the assertion $G_{m_2}^2$ within B_{m_2} , this implication shows Equation (8.2).

In this case, we can discard the assumption assertion A_{sw}^2 of the switching contract from B_{sys}^S . Let $B_{sys}^{S'}$ be the version of B_{sys}^S after we do so, i.e. $B_{sys}^S = B_{sys}^{S'} \cap A_{sw}^2$. Then we can focus on $B_{sys}^{S'}$ in consistency assessment. We already showed in Subsection 6.2.3 and in Section 7.4 that C_{m_2} and C_{sw} are consistent when $act_{m_2} = ON$. In $B_{sys}^{S'}$, the only remaining common port between mode m_2 component and the switching component is act_{m_2} . Also, since a mode contract is historyless between activation periods, for every trajectory tr over the activation port act_M of a mode M , there is a trajectory tr' in B_M such that $pr(tr')_{act_{m_2}} = tr$. This shows that

$$\forall tr_{sw} \in B_{sw} : \exists tr_{m_2} \in B_{m_2} : pr_{act_{m_2}}(tr_{sw}) = pr_{act_{m_2}}(tr_{m_2})$$

and that C_{sys}^S is consistent when S is active.

Consistency of switching among modes

The switching between two sets of modes that can occur during the operation of a system are defined by the mode-switching contract C_{sw} . For C_{sys} to be consistent, we need to ensure that a switching between a pair of sets of modes that is defined by C_{sw} is also possible in C_{sys} and not lost due to inconsistency among mode and mode-switching contracts.

Given that S_1 and S_2 be two sets of modes, and B an assertion with alphabet $P_B \supseteq S_1 \cup S_2$, let $Switch(S_1, S_2, B)$ be a predicate as defined in Equation (8.3). $Switch(S_1, S_2, B)$ states that in B there is a trajectory where a switch from S_1 to S_2 occurs.

$$Switch(S_1, S_2, B) =$$

$$\exists(\tau, w) \in pr_{S_1 \cup S_2}(B), j < \langle \tau \rangle :$$

$$RANGE(w_j) = \{(act_i, x) | act_i \in S_1 \cup S_2 \wedge act_i \in S_1 \iff x = ON\} \wedge$$

$$RANGE(w_{j+1}) = \{(act_i, x) | act_i \in S_1 \cup S_2 \wedge act_i \in S_2 \iff x = ON\} \quad (8.3)$$

Then, formally we define the consistency criterion for switching as follows:

$$\forall S_1, S_2 \in 2^{P_{act}}, S_1 \neq S_2 : Switch(S_1, S_2, B_{sw}) \implies Switch(S_1, S_2, B_{sys}) \quad (8.4)$$

Informally the above predicate states that if there is a trajectory in C_{sw} that allows for a switch from S_1 to S_2 to occur then there has to be a trajectory in C_{sys} that does so as well. In order to check the above criterion, we can identify the pairs of sets of modes where the second set of modes is activated after the first in C_{sw} . Then, for each such pair, we need to inspect by simulation the consistency of C_{sys} in situations where switch occurs.

We can obtain such pairs of sets of modes allowed by the switching contract C_{sw} by model checking if C_{sw} is expressed in Uppaal. For this we can employ an additional

Figure 8.19: The Uppaal automaton to check if C_{sw} allows a switch from a set of modes set1 to another set set2.

```

1 bool check_set(int set){
2   if (set%2 != act_m1)
3     return false;
4   else if ((set/2)%2 != act_m2)
5     return false;
6   else
7     return true;
8 }

```

Listing 8.5: The declaration of check_set function used in Figure 8.19

Uppaal model shown in Figure 8.19, which is a Uppaal template that takes two integer parameters, set1 and set2, encoding two sets of modes. The function check_set(int set) returns true if only the given set of modes active and false otherwise. For our robot case with two modes, the declaration of check_set function can be as shown in Listing 8.5. Initiating the Uppaal template in Figure 8.19 with the encoding of two different mode sets and composing it with the Uppaal model of C_{sw} , we can check if the switching from the first set of modes to the second is possible by checking for the following query:

- $E \leadsto \text{MSS.end}$

where MSS is the name of the instantiated Uppaal template.

In order to find trajectories in $B_{C_{sw}}$ in which, for a pair of a sets of modes a switch occurs from the first set of modes to the other, one can simulate $B_{C_{sw}}$ in Uppaal or use witness traces generated by Uppaal. Let Φ be the state property for such a switch for a pair of sets of modes. In the above example MSS.end is such a property. Then, we can use the query $\forall \square \neg \Phi$ and inspect the witness traces generated by Uppaal.

Like we did in checking consistency for a mode set, we can ignore some modes while assessing consistency of overall system contract for a pair of set of modes. Let S_1 and S_2 be a pair of set of modes defined such that $S_1, S_2 \in 2^{P_{act}}$. We want to check if there are trajectories in B_{sys} where a switch from S_1 to S_2 occurs. This check can be performed on a version of the system contract where only modes in $S_1 \cup S_2$ are involved:

$$C_{sys}^{S_1 \cup S_2} = C_{sw} \otimes C_{mc}^{S_1 \cup S_2}$$

where $C_{mc}^{S_1 \cup S_2}$ is the composition of the modes in $S_1 \cup S_2$.

Figure 8.20: Conditions to disprove (a) or strengthen (b) consistency with respect to switching from mode-set S_1 to S_2 .

For a particular pair of set of modes (S_1, S_2) , investigating consistency can be performed in hybrid system simulations using system models for generating trajectories and the observer for $C_{sys}^{S_1 \cup S_2}$. Since the $C_{sys}^{S_1 \cup S_2}$ is composed of individual observers for assumption and guarantee assertions of modes in $S_1 \cup S_2$ and the mode-switching contract, we can assess the conformance of a given trajectory to individual assertions of mode and mode-switching contracts. Note that, while performing simulations we need to focus on points of the trajectories where a switch from S_1 to S_2 occurs.

Now, we explain how the consistency of an overall system contract with respect to switches from a set of modes to another, i.e. from S_1 to S_2 . Let L_1 and L_2 be conjunctions of two sets of individual assertions such that $L_1 \supseteq B_{sys}^{S_1}$ and $L_2 \supseteq B_{sys}^{S_2}$. Then, if for all the trajectories we can show that $L_1 \Rightarrow \neg L_2$ when (S_1, S_2) switch happens, this means that:

$$\neg(\exists(\tau, w) \in pr_{S_1 \cup S_2}(L_1 \wedge L_2), j < \langle \tau \rangle : \\ \text{RANGE}(w_j) = \{(act_i, x) | act_i \in S_1 \cup S_2 \wedge act_i \in S_1 \Rightarrow x = ON \wedge act_i \notin S_1 \Rightarrow x = OFF\} \wedge \\ \text{RANGE}(w_{j+1}) = \{(act_i, x) | act_i \in S_1 \cup S_2 \wedge act_i \in S_2 \Rightarrow x = ON \wedge act_i \notin S_2 \Rightarrow x = OFF\})$$

This is illustrated in Figure 8.20a where the time line of such a trace is shown. Showing this for a pair of L_1 and L_2 would show the overall system contract is inconsistent with respect to (S_1, S_2) switches.

In Figure 8.20b, a different time line of traces is shown, where at the point of (S_1, S_2) switch $L_1 \Rightarrow L_2$ holds. For a pair of L_1 and L_2 showing this property does not alone prove consistency of overall system contract with respect to (S_1, S_2) . However, it would strengthen the confidence on consistency and now L_2 in $B_{sys}^{S_1 \cup S_2}$ can be discarded while continuing with assessment.

Now, using the robot case, we demonstrate partial assessment of consistency of overall system contract when a switch occurs from a set of modes to another. On our robot

Figure 8.21: Conditions to check to show the consistency of switching between mode sets S_{m_2} and S_{m_1}

case, we assess the consistency of switching between modes m_2 and m_1 . For the sake of simplicity we assume that there are no bumps or junctions. This means that we change our assumption on labyrinth structure, replacing the assumption assertion $A_{m_2}^3$ of mode m_2 with an assertion $A_{m_2}^{3'}$ that defines just a straight line. Since we assume that there are no bumps, we can also ignore $A_{m_2}^5$ which defines the behaviour of robot body in the case of a bump.

To show the consistency when switching from m_2 to m_1 is trivial since the only assumption included in the contract of mode m_1 is $A_{m_1}^1$ on kinematics of robot body which is shared with the contract of mode m_2 and it is satisfied at the time of the switch. To show the consistency when switching from m_1 to m_2 is more complex. Our strategy is to use the maximum deviation of the robot with respect to the line at the point of an m_2 to m_1 switch. And then use the guarantees provided by the mode m_1 to show that at the point of the next m_1 to m_2 switch, assumptions of m_2 are satisfied.

In Figure 8.21, a time axis with a switch from S_{m_2} to S_{m_1} and one from S_{m_1} to S_{m_2} are shown. At the time of the switch from S_{m_2} to S_{m_1} , we intend to show that the behaviours defined by $B_{sys}^{S_{m_2}}$ implies the assertion L_{dev} . L_{dev} defines, at the time of the switch, a set of values for the robot's deviation from the line with respect to its angle with the line and to its line sensors' position on the line. Then at the time of the switch from S_{m_1} to S_{m_2} , we intend to show that, when combined with L_{dev} , the behaviours dictated by $B_{sys}^{S_{m_2}}$ implies that the assumptions of S_{m_2} are satisfied.

Our strategy in showing the consistency of switching between the modes m_1 and m_2 back and forth consists of the following two steps performed in simulation:

1. In simulation, find the most deviating angle and position of the robot at the of the switch from S_{m_2} to S_{m_1} , i.e. L_{dev} , that it can have within S_{m_2} . Utilize the $C_{sys}^{S_{m_2}}$ observer while doing this to ensure we stay in the environments and implementations defined by $C_{sys}^{S_{m_2}}$.
2. In simulation, find out if there can be a trajectory that violates $A_{sys}^{S_{m_2}}$ the next time S_{m_2} becomes active. Utilize the $C_{sys}^{S_{m_1}}$ observer while doing this to ensure we stay in the environments and implementations defined by $C_{sys}^{S_{m_1}}$.

In order to accomplish the above tasks in simulation we use the 20-sim model shown

in Figure 8.22. The model is composed of three main parts denoted by three green boxes. The bottom green box labelled by *Implementation Model* encloses the implementation models for user, mode switching, the modes, the robot and the maze. The implementation models are used to investigate the behaviours defined by the system contract. The top green box labelled by C_{sys} encloses the observer models and their connections that make up the overall system contract. The middle green box labelled by *System Ports* encloses all the ports of the system. The system ports are used to connect the individual 20-sim blocks in the *Implementation Model* and the overall system contract C_{sys} .

In the implementation model, the blocks *Robot* and *Maze* models the kinematics of the robot and the structure of the maze. The blocks M_1 and M_2 denotes the controller models for modes m_1 and m_2 of the robot. The outputs of the mode controllers are selected by the block *Select*, according to the activated mode output by the *Mode Switching* block, and fed into the block *Robot*. Finally, the block *User* generates the user input fed into the *Mode Switching* block. Note that this implementation model, i.e. the overall architecture and the individual model blocks, is constructed with the aim of showing the consistency of switchings between the modes m_1 and m_2 such that it does not have to reflect an actual implementation. Also, it is not complete, especially with respect to the switching behaviour, in the sense that it is used to show consistency of the whole overall system contract.

In Figure 8.22, top green box enclosing the observer model for the overall system is fed with the values for the ports in the alphabet of the system contract. The values of the ports are assigned by the implementation models. The C_{sys} observer model generates a single output (C_{sys} output). C_{sys} observer model is composed of three observer models corresponding to observers for contracts C_{sw} , C_{m_1} and C_{m_2} . The outputs of these observers are combined in a way to obtain the output values for the assumption (A_{sys}) and guarantee (G_{sys}) assertions of the overall system contract.

We use the 20-sim model in Figure 8.22 to show consistency switching between the modes m_1 and m_2 . As depicted in Figure 8.21, for the first step, we intend to show that:

$$B_{sys}^{S_{m_2}} \Rightarrow L_{dev}$$

In order to show this, we run simulations which are started with various parameters for the robot's position on the line and angle to the line and where the robot steers from the line with, at the same time, staying in the behaviours defined by the contract for mode set S_{m_2} . We use 20-sim's optimization features to find the maximum deviation. In Figure 8.23 two plots resulting from the optimization for a maximum deviation are shown. In the top plot the robot's angle to the line and the outputs of the assumption (A_{m_2}) and guarantee (G_{m_2}) assertions of mode m_2 are shown. We omit in this plot the assertions of the switching contract, which is a part of the contract for the mode set S_{m_2} . All the assertions of the switching contract are satisfied during the shown simulation. The angle of the robot with the line in the maximum deviation is the value of the green line in the top plot where it crosses the pink line, where G_{m_2} goes

Figure 8.22: The complete 20-sim model used to assess consistency of C_{sys} .

Figure 8.23: Simulation of implementation models to find maximum deviation of robot from the line in S_{m_2} .

from true to false. This value indicates the maximum angle that robot can have with the line before invalidating G_{m_2} .

Note that the mode m_2 guarantees that the robot drives on the line with its speed in a certain interval (See Figure 6.12). In order to reach a maximum deviation the required speed is maintained while steering towards out of the line (towards left in the plot).

In the bottom plot of Figure 8.23, the values of the four line sensors in front of the robot are shown. The four lines in the plot named as lf_l0 , lf_l1 , lf_r0 and lf_r1 denote the leftmost sensor, the middle left sensor, the middle right sensor and the rightmost sensor, respectively. At the point where maximum deviation is reached the middle right sensor loses the line, where the middle left sensor had already lost the line. Indeed, the maximum deviation denotes the state of the robot where it has a maximum angle and about to leave the line, i.e. none of its middle sensors see the line. The assertion L_{dev} is defined such that the deviation of the robot cannot be more than the configuration plotted in Figure 8.23.

Note that m_1 does not assume anything about the robot's state at the time of activation. The only assumption in S_{m_1} regarding the ports to robot and the environment is $A_{m_1}^1$, which is the same as $A_{m_2}^4$ in S_{m_2} . Also, observe that the assumptions of S_{m_1} coming from the switching contract will be satisfied at the time of the switch since they are shared with S_{m_2} . Therefore, $B_{sys}^{S_{m_2}} \Rightarrow L_{dev} \wedge A_{sys}^{S_{m_2}}$ at the point of switch from S_{m_2} to S_{m_1} .

Figure 8.24: Simulation of implementation models to find the earliest time to get offline in mode set S_{m_1} when started with the robot at maximum deviation from the line.

As depicted in Figure 8.21, for the second step, we intend to show that:

$$L_{dev} \wedge B_{sys}^{S_{m_1}} \Rightarrow A_{sys}^{S_{m_2}}$$

The only guarantee of m_1 (and therefore of S_{m_1}) is $G_{m_1}^1$ which states that the robot should come to a halt within t_s time after activation of m_1 (See Subsection 6.2.2). The mode m_1 does guarantee anything about robot's behaviour between the mode activation and the time robot stops. Our intent is to investigate in simulation if, when started with the robot at a state of maximum deviation we determined previously, the mode set S_{m_1} allows robot to get into a state where the assumptions of the mode set S_{m_2} will be violated the next time a switch from S_{m_1} to S_{m_2} occurs. Observe that the assumptions of S_{m_2} coming from the switching contract will be satisfied at the time of the switch since they are shared with S_{m_1} . The assumption $A_{m_2}^2$ requires that the robot should be still at the time of m_2 activations. $A_{m_2}^2$ would be satisfied at the time of the switch from S_{m_1} to S_{m_2} since the time for the robot to stop in m_1 , i.e. T_s , is less than the minimum interval of m_1 activation defined in the guarantee, i.e. G_{sw} , of the switching contract: $T_s \leq \min(T_3, T_4)$. The assumption $A_{m_2}^3$ is satisfied at the time of the switch since we assume the labyrinth structure is preserved. The assumption $A_{m_2}^4$ is satisfied at the time of the switch since it is the same with $A_{m_1}^1$. The assumption $A_{m_2}^5$ is also satisfied at the time of the switch since we assume that there are no bumps.

We are left with the assumption $A_{m_2}^1$. $A_{m_2}^1$ states that the robot should see the line at the time of mode m_2 activation. This means that the port *offline* should be false at the time of activation. In simulation we can show that this is possible or not when switching from S_{m_1} to S_{m_2} given that at the time of activation of S_{m_1} the robot has

the previously determined maximum deviation from the line. In this direction, we obtained the earliest time after activation of S_{m_1} with maximum deviation that the robot's *offline* port becomes zero. In Figure 8.24 the associated plots are shown. In the top plot the point of switch from S_{m_1} to S_{m_2} is indicated. At the time of the switch the robot has the previously determined deviation from the line. After S_{m_1} is activated, the robot steers away from the line such that the port *offline* becomes true as soon as possible. To determine associated the parameters for controlling the robot for *offline* to become at the soonest time, we use 20-sim's optimization features. As seen on the second plot in Figure 8.24, after S_{m_1} is activated the angle of the robot with the line is changing and the robot becomes offline, i.e. it loses the line altogether, in about 50ms. However, since the guarantee assertion $G_{m_1}^1$ requires robot to come to an halt within $T_s = 40ms$ after activation, G_{m_1} goes becomes false before the robot becomes offline. This shows that switching from S_{m_2} to S_{m_1} and then again to S_{m_2} is consistent.

Note that with the two steps explained above, for the mode sets S_{m_1} and S_{m_2} and under the stated assumptions on the bumps and junctions, we showed not only the consistency of the overall system contract, which requires just existence of some behaviour (see Equation (8.4)), but a stronger property. We actually showed that switching from S_{m_2} to S_{m_1} cannot violate assumptions of S_{m_1} , and, following that, switching from S_{m_1} back to S_{m_2} cannot violate assumptions of S_{m_2} . This implies that implementations of C_{m_1} and C_{m_2} can be developed independently from each other.

8.5 Conclusions

In this chapter we presented the methods to construct and validate the overall system contract. Overall system contract is the composition of mode contracts and the switching contract. The assertions in the constituent contracts of the overall system contract can be expressed in 20-sim as hybrid observers or in Uppaal as non-observer specifications. In order to be able to execute both sorts of assertions together, i.e. the whole system contract, we present in Section 8.2 a method to execute Uppaal specifications in 20-sim. The method employs the model-based testing extension of Uppaal called Tron. We described how an assertion expressed in Uppaal can be supplemented with helper processes that enables using the assertion as an observer. Similarly, in 20-sim side, we describe how one can model an observer that makes external DLL calls to send variable values and delays to Tron tool to check if the sent values conforms to the associated Uppaal model. We described how we implemented the connection between Tron and 20-sim.

Regarding contract validation, we presented the correctness, completeness and consistency criteria adapted for an overall system contract. In particular, we extended the consistency criterion to cover for the consistency of individual mode sets and the switches between the mode sets. We provided the intuition and formal definitions of these extensions. On the robot case, we demonstrated how the extended consistency criterion can be checked on a system contract. We showed that the assume-guarantee nature and the executability of overall system contract enable reliably validating the consistency using a hybrid system simulator, 20-sim in our case.

After constructing and validating the overall system contract, the system specification is completed. The development process proceeds with system design. Aim of the system design process is to realize the system contract by successively refining it with design-oriented contracts and, in the end, implementing it with actual components that represent the execution platform and generated or hand-coded implementations. In the design process, the exact and executable nature of the specification contract and its formulation as an assume/guarantee pair(s) enable effective evaluation of conformance of the design contracts and models.

Part III

Design

9

Contract-Based System Design

In Part II, we described how to specify a control system using contracts. We adopted a mode-based architecture to structure the specification and expressed the specification as an observer-based A/G contract. The aim of this chapter is to describe how to proceed with the system design in order to realize the system specification contract. The initial design step is to perform a digital control design that conforms to the system specification. In Section 9.1, we present a formulation for the design contract that refines the specification contract. The design contract is based on a digital control design that conforms to the system specification. We describe how such a digital control design is performed in 20-sim simulation tool. In 20-sim, the digital control design is guided by the observer-based contracts for the individual modes, the switching logic and the overall system contract.

Once the digital control design is completed, the design process continues with implementation. On this part, in Subsection 9.1.3, we describe how to utilize vertical contracts to map a digital control contract to an execution platform. Such a vertical contract can be used to validate if the control system can be faithfully implemented on a particular execution platform.

9.1 System design

In this section we sketch out a digital control design process that realizes a specification contract. The system specification is expressed as an A/G contract $C_{sys} = (A_{sys}, G_{sys})$ which, as we have seen in Chapter 4, is a refinement of the initial system requirements:

$$C_{sys} \leq R_{sys}$$

The initial, informal system requirements R_{sys} are grouped into main system functions (system modes) and switching among them. This structure is reflected in the specification contract C_{sys} ; C_{sys} is the composition of mode and mode-switching contracts.

Once the specification is performed and validated as described in Part II, the digital control design is performed. Digital control design is expressed by the design contract C_{des} which refines the specification contract:

$$C_{des} \leq C_{sys} \tag{9.1}$$

Figure 9.1: System decomposition for digital control design. System is decomposed into control software, and sensing and actuating components.

Recall from Subsection 4.1.1 that the system specification is defined on the sets of variables and ports I_{Os} , O_{Os} and V_e where

- I_{Os} and O_{Os} are the ports from/to the existing subsystems where the system to be developed will be connected to.
- V_e is the set of variables of interest in the environment.

The variables in V_e are the variables that the system's desired function is defined on. Most of the time, however, these variables cannot be directly sensed/controlled. A digital control design achieves its control objectives by sensing/controlling the environment variables via sensors/actuators whose behaviors are described as mappings between the environment variables and the ports of the digital control system. Selecting the appropriate sensors/actuators is a part of the design process which the subsequent digital control design depends on. Figure 9.1 illustrates the system design as a composition of the digital control component and the component for sensors and actuators.

The digital control design, which is expressed by the control software component in Figure 9.1, is the process of designing the control laws to realize the system modes and designing the switching behavior. Note that the design is independent of the particular mode-based structure of the system specification, in that the designed control laws and the switching behavior do not have to reflect the modes and the switching behavior in specification. For instance, two different modes can be realized by the same control law in design with different parameters which are set by the switching behavior in design according to the switching behavior in specification.

Control design can be performed using contracts, like we do for system specification. In such a case, contracts for individual components for control laws and physical processes are formed using hybrid automata as the formalism to express assertions, and they are composed using the contract algebra resulting in C_{des} (Benvenuti et al., 2008a,b; Nuzzo et al., 2014, 2015). The challenge in this approach is that one has to mostly rely on formal verification on hybrid systems in order to ensure consistency of C_{des} and refinement of C_{sys} by C_{des} .

The alternative is to adopt a conventional control design process. The advantage of this is that the methods and tools used in conventional control design is very mature.

However, the control design process should still be integrated in the contract-based development: the control design should be guided by the specification contract and the result of the control design should be expressed as a contract. In this thesis we take this path. In such an approach, the digital control design is performed in a hybrid system simulation tool, such as Simulink or 20-sim. Let M_{cont} and M_{sa} be the models designed for the control software component and the component for sensors and actuators in Figure 9.1. Note that M_{cont} and M_{sa} are not contracts, i.e. pairs of assertions, but component models, i.e. individual assertions (Section 2.2). The modelling of M_{cont} and M_{sa} is performed such that, in combination, they are an implementation of the specification contract C_{sys} :

$$M_{cont} \wedge M_{sa} \models C_{sys}$$

which is valid if and only if:

$$A_{sys} \wedge (M_{cont} \wedge M_{sa}) \subseteq G_{sys} \quad (9.2)$$

Using M_{cont} and M_{sa} we can formulate the design contract C_{des} as:

$$C_{des} = (A_{sys}, M_{cont} \wedge M_{sa})$$

Note that Equation (9.2) implies that the refinement in Equation (9.1) since $A_{des} \supseteq G_{sys}$ and $G_{des} \subseteq A_{sys}$ in saturated form.

9.1.1 Digital Control Design

In the control design process, the aim is to obtain the digital control model M_{cont} and the model for sensors and actuators M_{sa} such that they, in combination, implement the system specification contract C_{sys} . In this subsection we briefly describe, for a given an observer-based system specification contract C_{sys} , how one can construct the digital control model M_{cont} and assess its validity.

In Figure 9.2, a 20-sim model that can be used to implement the specification contract for our robot case is shown. In the figure, the parts that make the M_{cont} , M_{sa} and C_{sys} are indicated. In the area marked as *control software model*, a sample digital control software model, i.e. M_{cont} , is shown. M_{cont} communicates with the user directly and with the robot through the sensors and actuators modelled by M_{sa} . M_{cont} is composed of a switch block, three control law blocks and a select block. The switch block communicates with the user, with control law blocks and with the sensors. The switch block determines which control laws are active at any time. Note that, in this example model we included three control laws whereas in the specification contract we had only two modes. This is to indicate that the architecture for the design is independent of the architecture of the specification.

In the area marked as *sensors and actuator model*, M_{sa} is shown. M_{sa} models the communication with M_{cont} and the robot block modelling robot dynamics. The robot block interacts with the maze represented by another block.

In the area marked as *system specification contract*, the observer blocks A_{sys} and G_{sys} modelling assumptions and guarantees of the specification contract C_{sys} are shown.

A_{sys} and G_{sys} are connected to the ports in P_{sys} on which the system specification contract C_{sys} is defined. In Figure 9.2, M_{cont} models a digital control system, so it is composed of discrete-time blocks which are periodically executed according to sampling rates. The rest of the blocks in the figure are continuous-time blocks, modelling continuous behaviours.

The observer blocks A_{sys} and G_{sys} in Figure 9.2, are used to assess whether the system design modelled by M_{cont} and M_{sa} implement the specification contract C_{sys} which is formulated in Equation (9.2). In order for Equation (9.2), and consequently Equation (9.1), to be satisfied, during simulations, the boolean outputs of observer models should satisfy the predicate $A_{sys} \implies G_{sys}$.

As we stated in Section 2.1, a caveat of using simulation is that it cannot handle non-determinism, which causes a problem in a digital control design process performed using observers. Recall that, C_{sys} reflects a mode-based specification architecture in that it involves ports and variables (P^{sw} in Subsection 6.1.1) between the mode-switching component and the mode-composition component. As stated before, system specification should be defined on the external ports of the system and the system design should be based on this black-box view. As a result specification and design architectures should be independent. Unfortunately, having internal ports in the specification contract C_{sys} requires those ports to be included in system design simulations, which inconveniently ties the design architecture to specification architecture. The design architecture can still be independent from the specification architecture to a certain point which allows the values for the internal ports to be generated from the variables/ports of system design models, i.e. M_{cont} and M_{sa} . This is depicted in Figure 9.2 using a separate block to generate the values for the internal ports P_{sw} of the specification contract C_{sys} .

That a design contract refines a specification contract is not sufficient for the design to be valid. It also needs to describe a deadlock-free reactive system. This requirement translates to our case, in which we have implementation models defined as hybrid assertions, as the implementations should not have any blocking executions. Recall that a blocking execution is defined as a finite and maximal execution in a set of hybrid executions, i.e. a hybrid assertion (Section 5.2).

In Chapter 8, we described the criteria for consistency of the system specification contract and the methods to assess consistency. Consistency of system contract implies existence of consistent implementations. However, it does not guarantee that any implementation will be consistent. Therefore, the system model, i.e. $M_{cont} \wedge M_{sa}$, should be assessed against any blocking executions through simulations.

9.1.2 Digital Control Model Semantics

In Subsection 9.1.1 we stated that the digital control model M_{cont} in Figure 9.2 is composed of discrete-time blocks which are periodically executed according to specified sampling rates. In this subsection we explain the semantics of such discrete-time models, hence of M_{cont} . We use the execution semantics of M_{cont} in Chapter 10 when we define vertical contracts to analyse schedulability of the digital control design.

Figure 9.2: Digital control design models in 20-sim

The semantics behind the execution of discrete-time models in most control design tools such as Simulink (The Mathworks Inc., 2016a), SCADE (Esterel Technologies, 2016) and 20-sim is *SBD* (Tripakis, 2008). The following description of *SBD* semantics is taken from (Tripakis, 2008). A *SBD* model is a directed graph of atomic computation blocks where blocks connected to each other by directed signals. An example *SBD* model is shown in Figure 9.3. A *SBD* model can be hierarchical in the sense that atomic blocks can be combined to define composite blocks. A *SBD* is called *flat* if it only contains atomic blocks and any hierarchical *SBD* can be made flat by flattening, i.e. successively replacing hierarchical blocks by their internal diagrams.

Each atomic block is either *combinational* (stateless) or *sequential* (having internal state). A sequential block can be further classified as *Moore-sequential* whose outputs only depends on the current state but not on the current outputs. A flat *SBD* is called *acyclic* if every cycle in the diagram involves at least one Moore-sequential block which guarantees that there is no cyclic dependency of the signals for a particular synchronous instant.

The semantics of a flat, acyclic *SBDs* are as follows. Each signal x in a diagram is a total function $x : N \rightarrow V_x$ where N denotes the set of synchronous instants: $N = \{0, 1, 2, 3, \dots\}$ and V_x is the domain for x . $x(k)$ denotes the value of signal x at time instant k . x can be an input whose value is determined by the environment, otherwise it is determined by the (unique) block that produces x . The important property of acyclic *SBDs* is that for an acyclic *SBD* there is a well-defined order of executing the blocks to compute values for all the signals in the diagram for a particular synchronous instant. In Figure 9.3, a simple *SBD* with four blocks is shown. Note that the block B has an input above which

Figure 9.3: A simple synchronous block diagram (SBD)

has a different notation, i.e. without an arrow, than the other inputs. Such an input is called a *trigger*, which must be of type Boolean. A trigger is used to trigger execution of a block only when the trigger has the boolean true value. For a signal x originating from a triggered block T , the semantics are $x(k) = T(k)$ if the trigger for T is true at instant k and $x(k) = T(k - 1)$ otherwise.

The model for digital control design M_{cont} in our design contract can be represented as a periodic execution of an acyclic SBD model. Using triggers for triggering execution of individual blocks, a SBD model with a base rate for periodic execution can represent a multi-rate digital control model with rates defined as multiples of the base rate.

In principle, execution of a SBD with a sample rate can be expressed as a hybrid assertion which is required since the selected formalism for our contracts is hybrid assertions. However, we use the SBD formalism to work on M_{cont} since, as we see in Subsection 9.1.3, it is more suitable for working with vertical contracts for addressing implementation concerns.

Generating code for an acyclic SBD model as a single, monolithic sequential code block is straightforward. This is due to fact that an acyclic SBD model can be topologically be sorted to generate sequential code. The only implementation concern in this case is that the worst-case execution time of the code block should be less than the specified sample time for the model. 20-sim can only generate code as a single block which is to be executed at a fixed sample rate on a single thread. However, some other tools, such as Simulink, are capable of generating code for single or multi-rate models which can be scheduled and executed on multi-threaded or multi-processor platforms. See (Tripakis, 2008) for a discussion of the methods for modular code generation from SBD models.

As it is in general with most of the synchronous models of computations, in SBDs, there is usually the zero execution time (ZET) assumption. ZET assumption states that the execution time of the computational tasks are zero so that, at any synchronous instant, the input signals are sampled and the output signals are produced at the same instant without any delays. Designing a control system with ZET assumption is handy since standard results from control theory can be used (Derler et al., 2013) However, implementing a system with ZET assumption in a strict sense is impossible, since computation always takes time.

The deadline execution time (DET) assumption relaxes the ZET assumption such that the execution is allowed take an amount of time d which is equal to or less than the sample time. Under the DET assumption, the inputs are sampled at the beginning of the sample period and the outputs have to be computed and written any time before a d amount of time passes after the beginning of the period. As opposed to ZET, a system with DET assumptions can be implementable with enough computation power. However, designing a control system with DET assumptions raises difficulties for the control engineers due to the non-determinism involved in regarding the precise times that the outputs are written. An implementable and deterministic solution is to use the logical execution time (LET) assumption. Under the LET assumption, the inputs are sampled at the beginning of a sample period, the outputs are computed before an l amount of time passes after the beginning of the period but the outputs are written precisely at the end of l . In (Derler et al., 2013), there is a proposal of using design contracts that specify the execution time agreements between control and software engineers.

The control design model M_{cont} can have ZET, DET, or LET assumption with the associated sample times and deadlines. In control design tools such as Simulink and 20-sim, the execution time assumption and the related parameters can be set accordingly. As we will explain in Chapter 10, for analysing schedulability of an implementation of particular control design, M_{cont} needs to have either a DET or LET assumption which set deadlines for execution of the control tasks.

9.1.3 Vertical Contracts

The mode and mode-switching contracts we formulated and composed to obtain the overall system specification contract are *horizontal contracts* in the sense that they are all at the same abstraction level. Horizontal contracts are useful in assessing consistency of the components that are at the same abstraction level. On the other hand, when a contract for a component at an abstraction level needs to be related to a contract of a component at a different abstraction level *vertical contracts* are used. Such a relation is needed when a specification platform at level $l + 1$ is mapped to an implementation platform at level l ; the components at the level $l + 1$ are implemented by the components at level l . This sort of mapping using vertical contracts are at the core of platform-based design with contracts (Nuzzo et al., 2015; Sangiovanni-Vincentelli et al., 2012).

We use a simple example to demonstrate the use of vertical contracts for mapping an application to an implementation platform. We only explain the notion of vertical contracts whose application in a formal setting, i.e. on a concrete formalism for contracts, can be found in (Benveniste et al., 2012) by which this explanation is inspired.

Let C be a contract of an application with architecture shown in Figure 9.4a. C is a composition of three subcontracts representing the three components M_1, M_2 and M_3 in the figure. Let P be contract for an execution platform with architecture shown in Figure 9.4b. P is a composition of two execution units and a communication bus shown in the figure. The contracts of the execution units involve operating system services and runnables corresponding to the individual components of C .

After we obtain the application contract C either by formulating all subcontracts from scratch or by composing pre-existing contracts or by doing a combination of the two, and we validate it, we need to map it onto the execution platform. This mapping refines C by its potential implementation. The mapping is represented by the conjunction of the application and platform contract $C \wedge P$. This conjunction requires a mapping between the variables/ports of C and P which is done via *synchronization tuples*. In a synchronization tuple, values and occurrences of different elements are unified. For instance, via a synchronization tuple, a variable in C can be unified to a variable in P .

For a pair of contracts C and P with architectures shown in Figure 9.4a and Figure 9.4b, a mapping can be done as follows. The communication between the components M_1 and M_2 in C can be mapped to a variable in left execution unit in Figure 9.4b. The two input ports of the C can be mapped to the associated events and messages on the input port of P which is connected to the communication bus. Similarly, the two output ports of the C can be mapped to the associated events and messages on the output port of P which is connected to the communication bus.

The vertical contract $C \wedge P$ by the definition of conjunction is a refinement of C . However, there still can be inconsistencies in $C \wedge P$, which needs to be checked to ensure the design can be implemented. Methods to check such inconsistencies are explained in (Nuzzo et al., 2015).

As stated in Subsection 9.1.1, the digital control design, which is expressed by the assertion M_{cont} in the design contract C_{des} , can be represented as a periodic execution of a flat and acyclic SBD model. In the next chapter (Chapter 10), we explain how to formulate a vertical contract that defines the mapping such a SBD model on an execution platform and analyse the consistency of it by performing schedulability analysis using model-checking on timed-CSP models.

9.2 Conclusions

In this chapter we described, for a given system specification contract, how to proceed with the system design to realize the system specification. The first step in the design process is digital control design in which a control design model, employing a control design tool, which implements the specification contract is developed. The control design model consists of two parts: a model M_{cont} that defines the digital control logic which can be implemented on an embedded computer and a model M_{sa} that defines the behaviours of sensors and actuators. Implementation relationship is defined such that the control design model should conform to the guarantees of the specification contract when it is placed in an environment that conforms to the assumptions of the specification contract. While performing the control design in a simulation tool, we described how an observer-based specification contract can be effectively used to assess if the control model being developed implements the system specification.

The control model resulting from the control design process can be used to construct an A/G design contract where the assumptions are the same with the assumptions of the specification contract and the guarantees are defined by the behaviours of the con-

Figure 9.4: Architectures of application contract C and platform contract P

trol model. We showed that such a design contract refines the specification contract and the contract-based design process can proceed on this design contract.

In this chapter, we also explained the notion of vertical contracts which are used to relate two contracts at successive abstraction levels. Vertical contracts can be used to map an application contract to an execution platform contract. In the next chapter we describe how we can analyse schedulability of a digital control implementation of design contract on an execution platform using vertical contracts.

10

Schedulability Analysis for Vertical Contracts

In this chapter we aim to do two things. First, in Section 10.1, we describe how we can formulate a vertical contract which maps a control design contract to a platform contract in a way to describe the scheduling and execution of the tasks in the control design on the execution platform. Second, in Section 10.2, we present a schedulability analysis framework based on model checking on timed-CSP models. This section is based on the work presented in (Oguz et al., 2012). The presented schedulability analysis framework can be used to check consistency of the vertical contract described in Section 10.1 with respect to execution and schedulability of the controller tasks in the design contract. However, the schedulability framework is generic such that it can be used also within a development process that does not employ contracts. Therefore, in Section 10.2, we explain the proposed framework in a generic way staying loyal to the original work.

10.1 Formulating Vertical Contracts for Schedulability

In Section 9.1 we described how we formulate a design contract C_{des} using a digital control model M_{cont} designed in a control design tool such as 20-sim. In Subsection 9.1.2, we stated that the control design model M_{cont} can be expressed as a flat and acyclic SBD model with a base sample time with an execution time assumption such as ZET, DET or LET. In Subsection 9.1.3, we defined notion of vertical contracts and sketched out how to use vertical contracts to map an application contract onto an execution platform contract. Building on these results, in this section, we sketch out how to formulate a vertical contract to specify the timing concerns of implementing a design contract C_{des} . The idea is to formulate an execution platform contract C_{plat} that guarantees scheduling and execution of controller tasks according to a scheduling scheme. After mapping M_{cont} onto C_{plat} we can have the assertion that describes the timing and order of the execution of the blocks in M_{cont} on the platform described by C_{plat} . In Section 10.2, we will show that we can formulate this assertion using timed CSP and we can perform the schedulability analysis via model checking in the PAT tool to ensure that the vertical contract $C_{des} \wedge C_{plat}$ is consistent.

In Chapter 9 we formulated the design contract C_{des} as

$$C_{des} = (A_{sys}, M_{cont} \wedge M_{sa})$$

Figure 10.1: A SBD model with two macro blocks and a switch block triggering the macro blocks

where M_{cont} is a model representing a digital control design such that the design contract refines the specification contract. As stated in Subsection 9.1.1, M_{cont} can be expressed as a flat and acyclic SBD model with a base sample time and a execution time assumption which can be ZET, DET or LET. Here we assume that M_{cont} has a particular structure as shown in Figure 10.1. We assume that M_{cont} has a single switch block (the block A in Figure 10.1) and a number of independent macro blocks. The switch block triggers the macro blocks. This structure is similar to the mode-based architecture of our specification contract, however, note that M_{cont} represents the design so the switching conditions and the macro blocks do not have to reflect the specification modes. We also assume that the macro blocks in M_{cont} have fixed sample times and associated execution deadlines which are smaller (DET) or equal (LET) to the sample times.

We formulate a platform contract as $C_{plat} = (T, G_{plat})$ where G_{plat} is the assertion that models the scheduling and execution of tasks. The architecture of an example C_{plat} with 2 execution units is shown in Figure 10.2. The input ports in the figure are used for the arrival events of the new tasks attributed with the values for task priority and best and worst case execution time estimates. The output ports are used for the events signalling task completions. The guarantee assertion G_{plat} of the platform asserts that the task events arriving at the input ports are scheduled and executed according to a particular scheduling scheme.

With M_{cont} and G_{plat} with the proposed architectures we can have the conjunction $M_{cont} \wedge G_{plat}$ with the following mapping. Input port of each atomic block within the macro blocks in M_{cont} is mapped to an input port of a execution unit attributed with the trigger for its macro block, a priority value, the BCET and WCET estimates and a unique id for the atomic block. The output ports of each atomic block in M_{cont} is mapped to the output port of the execution unit to which its input port is mapped.

Figure 10.2: Architecture of a platform contract with two CPUs.

With the described mapping we can have the conjunction $M_{cont} \wedge G_{plat}$ which enables us to have the vertical contract as

$$C_{des} \wedge C_{plat} = (A_{sys}, M_{cont} \wedge G_{plat} \wedge M_{sa})$$

We can check the consistency of this contract by checking the consistency of $M_{cont} \wedge G_{plat}$. The consistency of $M_{cont} \wedge G_{plat}$ as we formulated it means that the application assertion M_{cont} is schedulable on G_{plat} : the conjunction refines the application assertion M_{cont} and it satisfies the deadlines stated in M_{cont} at all times. In the next section we present a schedulability analysis framework based on timed CSP which enables us to formulate M_{cont} and G_{plat} in timed CSP and check the schedulability of the conjunction.

10.2 Schedulability Analysis with Timed-CSP Models

In this section we provide a schedulability analysis framework based on timed CSP. The work presented in this section is based on (Oguz et al., 2012). The schedulability analysis framework can be used to check the consistency of the vertical contract we described in Section 10.1. For this the conjunction $M_{cont} \wedge G_{plat}$ can be represented in timed CSP and the schedulability can be analysed with the framework we explain. However, the schedulability framework can also be used without a contract based approach. It can be used to analyse schedulability of control systems modelled in CSP and implemented using a library that supports CSP. This is how the original work in (Oguz et al., 2012) was formulated. Therefore, we present this section staying loyal to the original work.

10.2.1 Timing Concerns of Systems Specified by CSP

The timed interpretations of CSP assume that there are always sufficient resources for processes to execute. This is called the *maximal parallelism* assumption (Schneider, 1999, Chapter 9). With the maximal parallelism assumption, the implementation details such as resource limitations and the scheduling policies to resolve those

limitations are abstracted away. However, practical CSP implementations face actual resource limitations. To have real-time support, they need scheduling of processes to ensure orderly and efficient use of resources (Barnes and Welch, 2003). For instance, *occam 2* (SGS-Thomson Microelectronics, 1995) has fixed-priority scheduling of processes. Priorities are defined relatively by using asymmetric parallel `PRI PAR` and asymmetric alternative `PRI ALT` constructs. CSP libraries CTJ (Broenink et al., 1999), CT++ (Orlic and Broenink, 2004) and C++CSP2 (Brown, 2007) also have fixed-priority scheduling. These follow the tradition of *occam* to specify priorities relatively; but these support more levels of priority. With these libraries, it is possible to apply efficient fixed-priority scheduling solutions such as rate-monotonic (RM) scheduling. *occam-pi/KRoC* (Welch and Barnes, 2005) and LUNA (Bezemer et al., 2011) support absolute and time-varying priorities. With these frameworks, applying earliest-deadline-first (EDF) scheduling is also possible as well as RM scheduling. The Toc (Korsgaard and Hendseth, 2008) programming language directly supports EDF scheduling by fully integrating deadlines in process specifications.

For a CSP model of an embedded control system, schedulability analysis can be used to guarantee timeliness of the system. Performed on the model level, it provides a way to check whether all the time-constrained executions finish before their deadlines. Schedulability of the embedded control system should be analysed with respect to its particular execution platform and the associated scheduling scheme. It is required that there exists reliable execution time measurements for the tasks and that the variations in the execution times are accounted for in the analysis.

Here, we propose a model-based schedulability analysis framework for analysing CSP-oriented designs. Our framework is based on Stateful Timed CSP (Sun et al., 2013, 2009b), which is an extension of timed CSP (Schneider, 1999), and analysis is performed using the PAT (Sun et al., 2009a) model checker.

The framework is composed of task and processor models defined as timed-CSP processes in the PAT tool. These framework structures are high-level abstractions that define the scheduling behaviour in the system. They are used to construct a timed-CSP model with scheduling behaviour by instrumenting an initial CSP model without scheduling. In the constructed model, the tasks are assigned with fixed absolute priorities and mapped to processors. Processor models model the non-preemptive scheduling of the tasks. Best and worst case execution times for the tasks are also incorporated to account for variations in the execution times of the tasks. Schedulability of the constructed model can be analysed through dense-time model checking using the PAT tool.

We also propose a schedulability analysis workflow describing the successive refinement steps to construct a timed-CSP model with scheduling from an initial untimed one without scheduling. The initial model is refined with timing and scheduling information by using the proposed schedulability framework. We show that, throughout the construction process, the safety properties of the initial model are preserved. On the resulting model we perform, as well as the schedulability analysis, verification of the liveness properties and deadlock freedom. We demonstrate our schedulability

analysis workflow on a case study of control software design for a two-wheeled mobile robot.

Contributions of this work are as follows:

- A schedulability analysis framework to analyse multiprocessor schedulability of CSP models with non-preemptive fixed-priority tasks with variable execution times.
- A schedulability analysis workflow that describes how to refine, using the proposed framework, an untimed CSP model without scheduling into a timed one with scheduling behaviour. On the resulting model, we perform dense-time model checking to analyse its schedulability and verify liveness properties and deadlock freedom.

Related Work

In classical real-time scheduling theory, there are well established traditional schedulability analysis methods to check schedulability guarantee for various preemptive or non-preemptive scheduling schemes with periodic and/or sporadic tasks with fixed or dynamic priorities (Buttazzo, 2004; Burns, 1994). However, traditional schedulability analysis methods are seriously challenged in case of multiprocessor systems. Furthermore, the allowed task structures are restricted to periodic or sporadic tasks with simple precedence relationships. These methods cannot handle systems modelled in CSP which involve complex task relationships and arrival patterns.

Motivated by the process-oriented programming language Toc (Korsgaard and Hendseth, 2008), Korsgaard et al. (Korsgaard and Hendseth, 2011) provide a framework for traditional schedulability analysis of malleable jobs of arbitrary parallel structure. The framework models the jobs with a SEQ/PAR-structure that is suitable to the process-oriented formalism, and multiprocessor scheduling is supported. However, structuring of jobs is limited to only SEQ/PAR constructs without involving any synchronization and the jobs are required to be independent. The framework is used to conduct pessimistic but sustainable schedulability tests.

In recent years, real-time model checking emerged as an alternative way of doing schedulability analysis. These methods employ dense-time model checking to verify schedulability of a system. With these methods, multiprocessor schedulability analysis is inherently possible. The task arrival patterns and task synchronizations are less restricted, promoting the applicability of the analysis. Also, the analysis results are less pessimistic and provide higher processor utilization since systems and task arrivals can be modelled in more detail.

The prominent formalism in model-based schedulability analysis is timed automata (Alur and Dill, 1994) or the task automata (Fersman et al., 2007), an extension of timed automata with tasks. Uppaal real-time model checker (Behrmann et al., 2004) is mostly employed in the implementations.

Uppaal-based schedulability analysis frameworks (David et al., 2010; Mikučionis et al., 2010) and the TIMES tool (Amnell et al., 2004) (only single-processor scheduling) model the scheduling problem with task and resource models that synchronize with

each other, and the related additional data structures that store task and resource data. Various preemptive and non-preemptive scheduling algorithms are supported. Complex task arrivals and task relationships can be modelled in Uppaal. Then the schedulability problem is formulated as a reachability property and real-time model checking is performed. However, schedulability problems with preemptable tasks that have variable execution times require stopwatch automata power on which reachability analysis is undecidable in the general case (Krcal et al., 2007). For this reason, exact analysis for such problems is not possible; instead over-approximation methods are employed (David et al., 2010; Amnell et al., 2004).

Regarding our problem, schedulability analysis of systems described in CSP can be done using one of the Uppaal-based solutions mentioned above. This requires transforming the CSP models into Uppaal systems in accordance with the adopted Uppaal-based schedulability framework. Then the schedulability of the system can be verified by reachability checking in Uppaal. In case of an unschedulability result, a reverse mapping from the witness trace generated by Uppaal to the CSP model should be provided to locate the cause of unschedulability in the CSP model. However, defining such a transformation and a reverse mapping would be a tedious approach.

Instead, our approach is based on dense-time model checking on Stateful Timed CSP (Sun et al., 2009b, 2013) models using the PAT tool (Sun et al., 2009a). Multi-processor scheduling is supported and the schedulability scheme is non-preemptive fixed-priority scheduling. We also present an associated design workflow in which the schedulability analysis is embedded in the process-oriented design of the system where a scheduled and timed-CSP model is constructed. On this model, besides the schedulability, it is also possible to formally verify other properties of the system, such as liveness properties and deadlock freedom, using the PAT tool.

Several theories of timed process algebra that include structures to define resources and scheduling behaviour, such as ASCR (Brémond-Grégoire and Lee, 1997) or PARS (Mousavi et al., 2004), have been proposed. In these process algebras, resource constrained processes and scheduling of them are included in the syntax and the semantics. Our approach does not propose a new process algebra nor does it extend the semantics of timed CSP that it is based on. The former approach results in more expressive frameworks and it enables syntactically more natural specification of various resources, resource demanding tasks and scheduling schemes within the system description. On the other hand, our framework can benefit from better and more mature tool support. Furthermore, it suits better to the well-established process-oriented design practices that use the CSP formalism.

10.2.2 Timed CSP and Process Analysis Toolkit (PAT)

We give a brief overview of the syntax and informal semantics of Stateful Timed CSP and of the features of PAT model checker. For details, see Sun et al. (Sun et al., 2013) and the PAT user manual (Sun et al., 2009a).

The CSP process algebra (Hoare, 1978) provides a process-oriented foundation to specify, design and analyse concurrent and interacting systems. To allow for timed

Figure 10.3: Schedulability analysis workflow.

reasoning, extensions to the basic theory are available, which provide discrete-time (Roscoe, 1997) and continuous-time semantics (Reed and Roscoe, 1988; Reed, 1990; Schneider, 1999). The model checking tools FDR (Gibson-Robinson et al., 2014) and PAT (Sun et al., 2009a) support automatic verification of timed-CSP models.

PAT's Real-Time System (RTS) module is based on Stateful Timed CSP, which is an extension of Timed CSP (Schneider, 1999). The process model constructs used in our study to define processes in PAT's RTS module are as follows:

$$\begin{aligned}
 P = & \text{Stop} \mid \text{Skip} \mid e \rightarrow P \mid e \twoheadrightarrow P \mid a\{\text{program}\} \rightarrow P \mid \text{ch!exp} \rightarrow P \\
 & \mid \text{ch?x} \rightarrow P \mid [b] P \mid \text{if}(b) \{P\} \text{ else } \{Q\} \mid P \square Q \mid P \sqcap Q \mid P \setminus X \\
 & \mid P; Q \mid P_1 \parallel \dots \parallel P_n \mid P_1 \parallel \dots \parallel P_n \mid \text{atomic}\{P\} \mid P \text{ timeout}[d] Q \\
 & \mid \text{Wait}[d] \mid P \text{ interrupt}[d] Q \mid P \text{ within}[d] \mid P \text{ deadline}[d]
 \end{aligned}$$

The process *Stop* idles and does not engage in any events. The process *Skip* successfully terminates, possibly after idling for some time. The process $e \rightarrow P$ is initially prepared to engage in the event e and, after performing the event, it behaves as P . Urgent event prefixing ($e \twoheadrightarrow P$) is used to define urgent events that cannot delay. An event is enabled if it is not blocked by any of the system processes and an urgent event must occur as soon as it is enabled. The process $a\{\text{program}\} \rightarrow P$ performs the data operation a executing *program* which may be a simple procedure updating data variables or a complicated sequential program.

$ch!exp \rightarrow P$ and $ch?x \rightarrow P$ denote the processes prefixed with the events for writing to and reading from the channel ch , respectively. A channel named ch of buffer size n is declared with the phrase: "*channel ch n;*". If buffer size is given zero then the channel is synchronous. Channel readings can be guarded with predicates ($ch?[x>0]x \rightarrow P$) or with expected values ($ch?1 \rightarrow P$).

$[b]P$ defines a guarded process; it is blocked until the guard condition b is satisfied and then proceeds as P . A conditional choice based on the condition b is written as $if(b) P \text{ else } Q$.

An external choice ($P \sqcap Q$) between two processes in initially ready to perform the events that either processes can engage in. The choice is resolved by the observation of the first visible event, in favour of the process that performs the event. The process $P \sqcap Q$ with an internal choice describes a non-deterministic choice between P and Q that is resolved internally by the process itself.

The notation $P \setminus X$ is used to encapsulate a set of events in X within a process. All of the events in X are made internal to the process. They are removed from the interface of the process; no other processes may engage in them.

$P; Q$ denotes the sequential composition of two processes. P starts first, and when it terminates, the control is immediately transferred to Q .

$P_1 || \dots || P_n$ denotes the parallel composition of a number of processes. Any event which appears in the interface of more than one process must involve all of the such processes whenever it occurs. All the involved processes must synchronize on the event by simultaneously engaging in it. In contrast, in the interleaving composition ($P_1 ||| \dots ||| P_n$), all processes perform their events completely independent of each other and do not interact on any event except the termination events.

A process P can be defined to be atomic ($atomic\{P\}$) to associate it with higher priority. If an atomic process has an enabled event, the event will have priority over all the events from the non-atomic processes in the system. No other non-atomic event may occur if there is an enabled atomic event. If multiple atomic processes are enabled, then they interleave each other. Furthermore, an enabled atomic event is urgent; it has to happen immediately.

The process $P \text{ within}[d]$ is restricted to react, by engaging an observable event, within d time units. The process $P \text{ deadline}[d]$ must terminate within d time units. If the time restrictions defined by *within* or *deadline* constructs are not met, then the system deadlocks.

We note that, time restricted actions dictated by urgent event prefixing, atomic processes, *within* or *deadline* can not be blocked and delayed beyond the restriction by the environment. So, it is wise to use such constructs for the actions that the environment is not supposed to engage in.

Regarding the data types, PAT supports global variables of integer, Boolean and integer arrays. It also supports importing and using user defined data types or external static methods, which can be implemented in C#. Calling an externally defined method (a static method or a method of a user defined object) named f with parameters $p_1, \dots, p_n$

is done using the phrase $call(f, p_1, \dots, p_n)$. Method calls and updating and referencing variables can be done in the event prefixes with data operation or in the predicates of conditional statements.

PAT's RTS (Real-Time System) module supports verification of assertions (defined with the keyword *#assert*) of different kinds on the defined processes. The supported assertions include classical refinement/equivalence relationships, Linear Temporal Logic (LTL) properties, reachability properties and deadlock freedom. Based on clock zone abstraction (Sun et al., 2013), PAT's RTS module can handle dense-time model checking.

10.2.3 Schedulability Analysis Workflow

In this section we present our schedulability analysis approach. Our approach is based on dense-time model checking on Stateful Timed CSP models. For the rest of the paper we will use the terms "Stateful Timed CSP process" and "timed CSP process" interchangeably.

Our approach consists of a schedulability analysis framework and an associated schedulability analysis workflow. The schedulability analysis framework consists of reusable task and CPU models described as timed-CSP processes. These framework structures are used to add scheduling behaviour to a CSP model of a control software.

The schedulability analysis workflow describes how to construct and analyse a timed and scheduled CSP process from an initial untimed and unscheduled CSP process. We call the initial process the PIP and the process constructed using it, the platform-specific process (PSP). The schedulability workflow is depicted in Figure 10.3. Initially, we assume that there is a PIP of the control system. The PIP is the process that models the untimed, unscheduled platform-independent behaviour of the control system. Ideally, the PIP would be a result of an initial design and analysis phase and it would satisfy all the safety specifications. We explain our assumptions on the PIP in Section 10.2.4.

As the first step of the workflow, based on the PIP, the PSP of the control system is constructed by incorporating platform-independent timing information, execution platform constraints and hardware mapping and priority information (see the top three parallelograms in Figure 10.3). The step-by-step description of the construction process is as follows:

1. **Instrument the PIP with platform-independent timing** (Section 10.2.5)
The PIP is refined into a timed process by incorporating platform-independent timing such as urgent events, cycle times for periodic processes, minimum inter-arrival times for sporadic events, timeout points, etc.
2. **Specify hardware mapping, priorities and execution times** (Section 10.2.6)
In this step the number of processor or processor cores present in the execution platform is specified. For the computational tasks contained in the PIP, the hardware mapping, fixed priority values and best and worst case execution times are specified.
3. **Add task and CPU processes** (Section 10.2.7)

The PIP is instrumented with task and CPU processes of the framework that, in combination, define the scheduling behaviour.

The resulting PSP defines the timed and scheduled behaviour of the control software tied to a particular execution platform. At the verification step (Figure 10.3), we perform dense-time model checking on the PSP to verify schedulability of the control system against the specified deadlines. Deadline violations are checked by introducing timeout points in the PSP with deadline values and performing model checking to see if any timeouts can occur (Section 10.2.8).

Additionally, at this step, the PSP is also model checked to verify deadlock freedom and liveness properties of the system. Regarding only the events present in the PIP, the PSP is a timewise trace refinement of the PIP; all the safety specifications, which hold for a PIP also hold for the corresponding PSP. This feature suggests that platform-independent untimed safety analysis can be done on the PIP and the results do carry over to any PSP constructed from this PIP. We show this refinement relation in the Appendix.

In case of any negative results at the verification step, the witness traces, if any, should be examined, and the PSP should be revised by modifying the execution platform constraints, priority assignments or hardware mapping.

10.2.4 Platform-Independent Process (PIP)

In this section we present our assumptions on the structure of a PIP. For the schedulability analysis, we assume there is a PIP that models the untimed, platform-independent behaviour of the control system. Even though timeliness is important for a control system, there is no timing yet in the PIP. The PIP only deals with the order of events and it is assumed to satisfy all the safety specifications of the system.

A typical control system is composed of a number of periodic and/or aperiodic control computations. Each of these control computations can involve a number of task executions such as control algorithm computations or processing of sensor data. In this direction, we assume that, in the control system that the PIP represents, there are a number of computational tasks.

In the PIP description we require that each computational task is represented by a single “task event” denoting the end of the task execution. Task events are not meant to be involved in any synchronization; they are actually placeholder events that are to be replaced by the task processes while constructing the PSP. Consequently, we require two kinds of events in the PIP process description. The first is the task event kind, as explained above. The second is the normal event kind. All the events (reading and writing to channels, synchronization events and data operation events, etc.) that are not specified as task events are normal events.

A PSP constructed from a PIP models the dispatching, scheduling and execution of computational tasks in timed CSP. These operations require tasks to be tracked with unique id’s. To ensure that a PSP is feasible to model check, we restrict the set of task events in a PIP to be fixed and finite with each task event given a unique name. We also require that, in a PIP, there is no interleaving or parallel combination of any two

```

1 //CP0 periodically computes a setpoint for CP1
2 CP0 = cp0_in -> task.0 -> write_setpoint ->
3   CP0;
4
5 //CP1 periodically reads a setpoint and computes a control output
6 CP1 = read_setpoint -> task.1 -> task.2 -> cp1_out ->
7   CP1;
8
9 SYSTEM = CP0 ||| CP1;

```

Listing 10.1: A sample platform-independent process (PIP) called `SYSTEM` including 3 task kinds.

processes which include the same task event so as to prevent overlapping dispatches of the same task in a PSP. These restrictions enable us to avoid online generation of unique task id's, an operation that would prohibitively increase the statespace of a PSP. The designer should ensure that a PIP obeys these restrictions. For complex models it can be hard to check the validity of a PIP manually, however, the validity check can easily be automated.

An example PIP named `SYSTEM` is shown in Listing 10.1. It is composed of two processes, `CP0` and `CP1`. `CP0` computes and writes a setpoint value which `CP1` reads but the communication is not synchronized since we would like the two processes to be executed at different frequencies. Actual transfer of the setpoint values are abstracted away in the model. Writing and reading the setpoint values are denoted by two distinct events, namely the `write_setpoint` and `read_setpoint` events. In `SYSTEM` there are three kinds of computational tasks which are denoted by the placeholder task events `task.0`, `task.1` and `task.2`.

10.2.5 Instrumenting the PIP with Platform-Independent Timing

In this section we explain the first step of constructing a PSP from a PIP. In this step, the PIP is refined into a timed process by incorporating platform-independent timing information such as urgent events, cycle times for periodic processes, minimum inter-arrival times for sporadic events, timeout points, etc. In order to specify such time-sensitive behaviour, the PIP is instrumented with timed operators of Stateful Timed CSP (`Wait[t]`, `Timeout[t]`, `Interrupt[t]`, `Deadline[t]`, `Within[t]` and the urgent event prefix `->>`).

All the events in the PIP that cannot be delayed are converted to urgent events. Hidden events, which are not of interest for verification, placeholder task events and the other events that can be delayed are left unmodified. The delay process `Wait[t]` can be inserted anywhere in the PIP description to introduce delays. Similarly, the timed operators `Deadline[t]` and `Within[t]` can be inserted anywhere to introduce time restricted behaviour. The untimed event interrupts and external choices can be replaced by timed interrupts (`Interrupt[t]`) and by timeouts (`Timeout[t]`), respectively.

```

1 CP0 = ((cp0_in ->> task.0 -> write_setpoint ->> Skip) ||| Wait[20]);
2   CP0;
3
4 CP1 = ((read_setpoint ->> task.1 -> task.2 -> cp1_out ->> Skip) ||| Wait[10]);
5   CP1;
6
7 SYSTEM = CP0 ||| CP1;

```

Listing 10.2: The process `SYSTEM` after it is converted into a timed process.

In Listing 10.2, the running example `SYSTEM` process is shown after it is converted to a timed process. All the events except the task events are converted to urgent events. Also, the processes `CP0` and `CP1` are modified to have periods of 20 and 10 time units respectively. This is done by first adding `Skip` processes in interleaving combination to the parts of the processes, and then replacing the `Skip`'s with `Wait` processes.

After this step the resulting process is a timed version of the PIP that is still platform-independent since there is no scheduling of the tasks and the execution times of the tasks are not incorporated into the model yet.

10.2.6 Specifying Hardware Mapping, Priorities and Execution Times

In order to perform the schedulability analysis for a particular execution platform, the PIP needs to be supplemented with execution times and priorities of the tasks and their mapping to the CPUs (or the CPU cores) present in the execution platform. In our schedulability framework, tasks are statically mapped to CPUs. To accommodate this, we adopt a task model, which specifies the following attributes for a task:

- `T_ID`: Unique id of the task.
- `BCET`: Best case execution time (≥ 1) of the task for the assigned CPU.
- `WCET`: Worst case execution time ($\geq \text{BCET}$) of the task for the assigned CPU.
- `PRIORITY`: Priority of the task. Higher the priority value, higher the priority of the task is.
- `CPU_ID`: Id of the CPU that the task is assigned to.

For each unique task event in the PIP, we need to specify its attributes. However, the tasks grouped into a process representing a component of the system usually share the same priority and might be assigned to the same CPU. So, most of the time, it is more convenient to assign a CPU and a priority value to a process that includes a number of tasks instead of doing that explicitly for each of the tasks. A process that is explicitly assigned to a particular CPU with a priority value is called a “mapped process”. Priority of a mapped process is defined according the selected priority assignment scheme. A process that is not explicitly assigned to a CPU inherits the mapping and the priority of its parent process. Depending on the design decisions, the selection of the mapped processes can be done in several ways as long as it unambiguously specifies a CPU and a priority value for each of the task events in the PIP.

```

1 //task_arr[t_id][attr]
2 //attr: 0 -> BCET
3 //      1 -> WCET
4 //      2 -> MP_ID
5 var task_arr[3][3] = {4,6,0, //t_id=0: task.0
6                       1,3,1, //      1: task.1
7                       1,3,1}; //      2: task.2
8
9 //mp_arr[mp_id][attr]
10 //attr: 0 -> PRIORITY
11 //      1 -> CPU_ID
12 var mp_arr[2][2] = {1,0, //mp_id=0: CP0
13                    2,0}; //      1: CP1

```

Listing 10.3: Sample Task and Mapped Process Attribute Arrays for `SYSTEM`.

A mapped process has the following attributes:

- `MP_ID`: Unique id of the mapped process.
- `PRIORITY`: Priority of the mapped process. Higher the value, higher the priority of the mapped process is.
- `CPU_ID`: Id of the CPU that the mapped process is assigned to.

In a PSP, attributes of all the mapped processes and the tasks are stored in two 2D arrays called `task_arr` and `mp_arr`. The arrays are indexed first by the unique id of the task or the mapped process and then by the attribute. In a task attribute array, for each task, we include an indirection to the mapped process that the task belongs to. Using this indirection, the priority and the mapping of a task are inherited from the mapped process containing the task.

In Listing 10.3, sample attribute arrays for the tasks and mapped processes defined for the PIP example `SYSTEM` are shown. The selected mapped processes are `CP0` and `CP1`, which are assigned to the same CPU with the id 0. `CP1` is assigned with higher priority.

Modifying an existing hardware mapping configuration (existing task and mapped process arrays) for a particular execution platform might require altering not only the mapped process array but also the task array. This is due to that, in the new configuration, the selection of mapped processes might be different. This may require updating the `MP_ID` fields of the task array. Additionally the `BCET` and `WCET` values for the tasks might be different for their new CPU assignments. However, given `BCET` and `WCET` values of the tasks for all kinds of CPUs in the execution platform, generation of mapped process and task arrays for a particular selection and mapping of mapped processes can be easily automated with a script.

10.2.7 Adding Task and CPU Processes

So far, we have refined the PIP into a timed process with platform-independent timing. We have specified hardware mapping, priorities and `BCET` and `WCET` of the tasks in task and mapped process attribute arrays. However, the modified PIP does not have

the scheduling of the tasks yet. The information specified in the task and mapped process attribute arrays is not yet used.

In this section we instrument the PIP process with schedulability framework structures which define the scheduling behaviour. Schedulability framework structures consist of the template processes for tasks and CPUs, related macro definitions and related channel and variable declarations (Listing 10.4). The template processes for tasks and CPUs are called `TASK` and `CPU`, respectively. The `TASK` and `CPU` processes are boilerplate models that are reused for every CSP model to be analysed.

Task process

A task process is an instance of the template process `TASK` (Line 14, Listing 10.4) parametrized by the unique id of the task. When enabled, an instance of `TASK` signals its release by the urgent event `push_task` that calls the macro `push` to add the task into the priority queue of the corresponding CPU. Then `TASK` waits to synchronize on the `finished_cpu[cpu_id]` channel with the corresponding CPU process; this communication event signals that the execution of the task is finished.

We use the `TASK` process to instrument the PIP: *each placeholder task event in the PIP process description is replaced by an instance of the `TASK` process*. The unique id of a replaced task event is given as the argument to the replacing `TASK` instance. In Listing 10.5, `CP0` and `CP1` processes of the PIP `SYSTEM` are shown after replacing all the placeholder task events with `TASK` process instances.

CPU process

In a PSP, scheduling and execution of the released tasks are modelled by processes that represent CPUs. Each CPU (or CPU core) present in the execution platform is represented by a separate instance of the template process `CPU` (Line 18, Listing 10.4). Instances of `CPU` are given unique ids from 0 to `_NUM_OF_CPUS-1` which is the total number of CPUs in the system.

In our framework, the associated scheduling policy is non-preemptive fixed priority scheduling. Each instance of `CPU` uses an external FIFO priority queue data structure called `cpu_queues` to sort the tasks according to their priorities. The priority queue data structure is implemented externally in C# and imported in the PAT tool. It has the `push`, `get_first`, `get_count` and `remove` operations which are defined as macros. `cpu_queues` and its operations are defined in lines 4-10 of Listing 10.4.

All the processes, variables, channels and queues used in the definition of the process `CPU` are parametrized with `cpu_id` of the corresponding CPU. An instance of `CPU` fetches tasks from its priority queue and, upon dispatching, delays for a certain time to simulate task executions. `CPU` is the interleaving combination of two sub-processes, `CPU_NEXT` and `CPU_EXEC`. The variable `exec_id` (Line 20, Listing 10.4) stores the id of the first task in the priority queue which is going to be executed next. Initially and after dispatching a task, `exec_id` is reset to -1.

`CPU_NEXT` (Lines 22 - 25) is responsible for assigning and, whenever necessary, refreshing the value of `exec_id` by polling the priority queue. It assigns `exec_id` with

```

1 channel dispatch_task[_NUM_OF_CPUS] 1;
2 channel finished_task[_NUM_OF_CPUS] 0;
3
4 var<PriorityQueueList> cpu_queues = new PriorityQueueList(_NUM_OF_CPUS);
5 #define push(cpu_id, t_id)
6   cpu_queues.Push(cpu_id, mp_arr[task_arr[t_id][2]][0], t_id);
7 #define get_first(cpu_id) cpu_queues.First(cpu_id);
8 #define get_count(cpu_id) cpu_queues.GetCount(cpu_id);
9 #define remove(cpu_id, t_id)
10   cpu_queues.Remove(cpu_id, mp_arr[task_arr[t_id][2]][0], t_id);
11
12 // Task process
13 #define task_cpu(t_id) mp_arr[task_arr[t_id][2]][1];
14 TASK(t_id) = push_task(call(push, call(task_cpu, t_id), t_id) ->>
15   finished_task(call(task_cpu, t_id))?{x==t_id}x ->> Skip;
16
17 // CPU process
18 CPU(cpu_id) = CPU_NEXT(cpu_id) ||| CPU_EXEC(cpu_id);
19
20 var exec_id[_NUM_OF_CPUS] = [-1(_NUM_OF_CPUS)];
21
22 CPU_NEXT(cpu_id) =
23   [call(get_count, cpu_id)>0 && exec_id[cpu_id] != call(get_first, cpu_id)]
24   assign_top{exec_id[cpu_id] = call(get_first, cpu_id)} ->>
25   CPU_NEXT(cpu_id);
26
27 CPU_EXEC(cpu_id) =
28   [exec_id[cpu_id] != -1]execute ->>
29   atomic{Wait[1];
30     dispatch_task[cpu_id]!exec_id[cpu_id] ->>
31     remove_task(call(remove, cpu_id, exec_id[cpu_id])) ->>
32     reset_exec{exec_id[cpu_id] = -1} ->>
33     Skip;
34   dispatch_task[cpu_id]?t_id ->> CPU_DELAY(t_id, cpu_id);
35   CPU_EXEC(cpu_id);
36
37 CPU_DELAY(t_id, cpu_id) =
38   (Wait[task_arr[t_id][0] - 1]; end_task -> Skip) within[task_arr[t_id][1] - 1];
39   finished_task[cpu_id]!t_id ->> Skip;
40
41 CPUS = ||| x : {0..(_NUM_OF_CPUS-1)} @ CPU(x);

```

Listing 10.4: Description of TASK and CPU processes.

```

1 CP0 = ((cp0_in ->> TASK(0); write_setpoint ->> Skip) ||| Wait[20]);
2   CP0;
3
4 CP1 = ((read_setpoint ->> TASK(1); TASK(2); cp1_out ->> Skip) ||| Wait[10]);
5   CP1;
6
7 PSP_SYSTEM = (CP0 ||| CP1) || CPUS;

```

Listing 10.5: SYSTEM process after instrumented with TASK and CPU processes.

the `t_id` of the first task in the priority queue whenever the queue is not empty and the value of `exec_id` is different from the `t_id` of the first task in the queue.

`CPU_EXEC` (Lines 27 - 35) defines the behaviour for dispatching and executing a task. Whenever the urgent event `execute` is enabled (when the CPU is idle and there is at least one released task in the queue), a task is dispatched at that time. The execution time of a dispatched task is represented by a delay for one time unit (with the `Wait[1]` process inside the atomic block), plus, in `CPU_DELAY` process, a delay for the interval `[BCET-1,WCET-1]`. Then, `CPU_DELAY` synchronizes on channel `finished_cpu[cpu_id]` with the `TASK` associated with the id of the dispatched task to signal the end of the task execution.

The task dispatch point is the start of the atomic block (Line 29, Listing 10.4). By the start of the atomic block (before the clock for `Wait[1]` starts ticking), the `Wait[1]` process forces all the urgent events in the PSP that might happen, to happen. This guarantees that all the task releases in the PSP that cannot delay happen and `exec_id` is assigned, in `CPU_NEXT`, with the first task in the priority queue.

The atomic block in `CPU_EXEC` has a critical function. Without the atomic block, value of `exec_id` might change due to a later task release before or just after the end of `Wait[1]` process causing a wrong value (for the `t_id` of the task to be dispatched) written to the buffered channel `dispatch_task`. We are abusing here the property that the enabled atomic events have higher priority than all the enabled non-atomic events. Given that the PSP does not include any other atomic blocks, the atomic block in `CPU_EXEC` ensures that no other event happens between the start of the atomic block, which is the activation of the clock for `Wait[1]`, and the end of the atomic block, which is the occurrence of `Skip`.

This atomic block seems to cause a time digitization such that an event of the PSP can either happen before the atomic block or after the `Skip` at the end of it which happens exactly after one time unit. However, this digitization effect does not remove any detectable behaviour. This is because, in PAT, one can only use integers inside the timed operators and it is not possible specify an event to happen strictly after or before a specified time.

In order to add to the PIP the scheduling behaviour defined by CPU processes we use a process called `CPUS` (Line 41, Listing 10.4). For each CPU present in the execution platform, a separate instance of `CPU` is initiated and all the `CPU` instances are put in interleaving combination to define the process `CPUS`. Then *CPUS is put in a parallel combination with the PIP which was instrumented with TASK processes*. For the running example `SYSTEM` process, the corresponding PSP called `PSP_SYSTEM` is shown in Listing 10.5 after the PIP is instrumented with `TASK` and `CPU` processes as the last step of the PSP construction. Note that instrumentation of the PIP with `TASK` and `CPU` processes can be automated once all the placeholder task events in the PIP process description are specified and enumerated.

```

1 CP0 = ((d_start.0 ->>
2     cp0_in ->> TASK(0); write_setpoint ->> d_end.0 ->> Skip)
3     ||| Wait(20));
4     CP0;
5
6 CP1 = ((d_start.1 ->>
7     read_setpoint ->> TASK(1); TASK(2); cp1_out ->> d_end.1 ->> Skip)
8     ||| Wait(10));
9     CP1;
10
11 PSP_SYSTEM = (CP0 ||| CP1) || CPUS || DEADLINES;

```

Listing 10.6: PSP_SYSTEM process after two time constrained processes in CP1 and CP1 are marked with `d_start.i` and `d_end.i` events.

```

1 #define _DEADLINE_COUNT 2;
2 var deadline_arr[_DEADLINE_COUNT] = [18,9];

```

Listing 10.7: Deadline count and values defined for PSP_SYSTEM.

10.2.8 Schedulability Analysis

After all the described construction steps are performed, the resulting PSP is a scheduled and timed-CSP model of the system. In this section, we explain how to perform schedulability analysis on a PSP. The purpose of schedulability analysis is to ensure that the control system (control software and execution platform) will satisfy its timing requirements. To verify the schedulability of the system, first we specify, on the PSP, a number of processes that, once started, have time constraints (deadlines) to finish.

The number of such time constrained processes are denoted by the macro definition `_DEADLINE_COUNT`. We instrument the PSP with such number of urgent event pairs `d_start.i` and `d_end.i` to mark the start and the end of the i^{th} time constrained process, respectively. The example PSP `PSP_SYSTEM` is shown in Listing 10.6 after two time constrained processes are specified in `CP0` and `CP1`.

The deadline values for the time constrained parts are specified in an array called `deadline_arr` that is indexed by the id of the time constrained part. A sample `deadline_arr` for `PSP_SYSTEM`, which specifies deadline values 18 and 9 for the two time constrained processes, is shown in Listing 10.7.

We need to verify that, for each of the time constrained processes, the associated deadline is not violated. For this purpose we use a deadline violation checking process called `DEADLINE(i, d)` for each time constrained process (Listing 10.8). `DEADLINE(i, d)` involves a timeout event `missed.i` denoting a deadline violation for the specified time constrained process (i) and specified deadline value (d); a `missed.i` event may only happen if the corresponding `d_end.i` event does not happen within d time units.


```

1 #alphabet DEADLINE {i:{0.._DEADLINE_COUNT-1} @ d_start.i, missed.i, d_end.i};
2 DEADLINE(i, d) = d_start.i ->>
3     (d_end.i ->> Skip timeout[d+1] missed.i ->> d_end.i ->> Skip);
4     DEADLINE(i, d);
5
6 DEADLINES = ||| i:{0.._DEADLINE_COUNT-1} @ DEADLINE(i, deadline_arr[i]);

```

Listing 10.8: Descriptions of DEADLINE and DEADLINES processes.

```

1 #assert PSP_SYSTEM |= []!(missed.0 || missed.1);
2
3 #assert PSP_SYSTEM deadlockfree;

```

Listing 10.9: Assertions for PSP_SYSTEM.

To verify the schedulability, we first formulate a process called DEADLINES (Listing 10.8) which is the combination of all DEADLINE processes. DEADLINES is put in a parallel combination with the PSP (see Listing 10.6). Then, we formulate and verify a schedulability assertion that says none of the `missed.i` events may occur in the PSP combined with DEADLINES process. If the verification of the schedulability assertion fails, then the witness traces should be examined to see which of the deadlines can be violated and the reasons. For the example PSP `PSP_SYSTEM`, the schedulability assertion (Listing 10.9) holds.

In addition to the schedulability assertion, we may also verify other system specifications on a PSP. A PSP is a trace timewise refinement of an initial PIP implying that, regarding only the events in the PIP, all the finite traces of the PSP are also included in the traces of the PIP. Any safety property that holds for the PIP is also valid for the PSP. So, only the remaining liveness specifications and deadlock freedom should be verified on the PSP (Listing 10.9). In case of any negative verification result, the witness traces, if any, should be examined to locate the cause.

When the verification of the schedulability assertion or any of the other system specifications fails, it can be caused by any of the parts (Figure 10.3) that were used to construct the PSP. It is left to the designer to locate the cause and revise the related part to make the PSP schedulable and satisfy all the system specifications.

10.2.9 Refinement Relation Between PIP and PSP

We would like to show that, if only the non-placeholder events in a PIP description are considered, there is a trace timewise refinement relation between a PIP and a corresponding PSP constructed from it. The idea is to divide the construction of a PSP from a PIP into successive steps and then to show that at each step the refinement relation is preserved.

In the following formulas we adopt the notation used by Schneider in (Schneider, 1999). Essentially we show that when we hide all the events in the PIP and the PSP

except the non-placeholder events in the alphabet of the PIP, then the PSP is a trace timewise refinement of the PIP. Let P_{PE} denote the process that results when all the events of a process P are hidden except the non-placeholder events for a particular PIP:

$$P_{PE} = P \setminus \{e : \alpha P \mid e \notin \alpha PIP \vee e \in ET\}$$

where ET denotes the set of placeholder task events in the PIP. Then we claim that, for a PIP and a corresponding PSP constructed from it, the following trace timewise refinement relation holds:

$$PIP_{PE} \sqsubseteq_{TF} PSP_{PE}$$

In a general sense, this kind of refinement relation between a PIP and a corresponding PSP can be demonstrated by dividing the construction of the PSP into successive steps and showing that the refinement relation is preserved at each step. To this end, the construction process can be divided into four steps, successively refining the PIP by adding scheduling and timing, and in the end obtaining a PSP :

$$PIP \rightarrow PIP' \rightarrow UPSP \rightarrow PSP$$

In the first step, a process PIP' is constructed by removing all occurrences of the placeholder task events from the PIP . Intuitively, PIP' is essentially the same process as PIP_{PE} , since none of the placeholder events in the PIP are involved in any synchronization; removing them is the same as hiding them, not changing the partial order of other events in the traces:

$$PIP_{PE} = PIP'$$

Second, we construct a process $UPSP$, an untimed version of the PSP , by inserting **TASK** processes in the places of previously removed placeholder events in PIP' , combining it with **CPUS**, including particular mapped process and task attribute arrays, and then converting the resulting process to an untimed CSP process. Conversion to untimed CSP is done by removing **Wait** and **Within** operators in **CPUS**, changing atomic blocks to normal blocks and converting urgent event prefixes to normal event prefixes.

$UPSP_{PE}$ cannot have any new traces that PIP' does not have. $UPSP$ preserves the process structure of PIP' regarding the events of PIP' ; it only includes additional **TASK** processes which do not interfere with the events or data that are common with PIP' . The scheduling of **TASK** processes alters only the partial order of the communication events `ready_cpu` and `finished_cpu` which are contained in the **TASK** processes.

However, the traces of $UPSP_{PE}$ cannot be any less than the traces of PIP' either. This is due to that, at any time, execution of enabled **TASK** processes (an execution of a **TASK** process is defined as a pair of `ready_cpu` and `finished_cpu` occurrences) can interleave each other. So $UPSP_{PE}$ is actually a trace equivalent of PIP' :

$$PIP_{PE} =_T UPSP_{PE}$$

As the last step, we add timing to *UPSP*. We convert the *TASK* and *CPU* processes back to their timed versions. We can introduce *Wait* processes into any point in *UPSP* to introduce delays. Similarly, we can introduce *Deadline* and *Within* operators at any point in *UPSP* to introduce time constrained behaviours. We can translate any external choice into a timeout choice and any event interrupt into a timed interrupt. We can convert normal event prefixes to urgent event prefixes and enclose some processes into atomic blocks.

The resulting process is *PSP* and we claim it is a trace timewise refinement of *UPSP*. We base our argument on Schneider (Schneider, 1999, pp. 397-403), according to which "all of the process operators preserve timewise refinement" and "timewise refinement process is compositional: a system may be refined by introducing timing information to each of its components independently."

Schneider only addresses timed CSP semantics and Stateful Timed CSP has extensions to it such as urgent event prefixes ($\rightarrow$), atomic blocks ($\text{atomic}\{\dots\}$), and *Within* and *Deadline* operators. All of these extensions impose time constrained behaviour. Additionally, the events in the atomic blocks have higher priority than all the other non-atomic events.

However, regarding the finite traces, the added time constrained behaviour and priority of some of the events cannot introduce any new traces; they can only restrict some of the traces. Therefore, we can make the following additions to the list in Schneider (Schneider, 1999, pp. 400) of unary CSP operators that preserve trace timewise refinement:

$$\begin{aligned}
 &\text{if } P_T \sqsubseteq_{TF} Q \text{ then} \\
 &\quad a \rightarrow P_T \sqsubseteq_{TF} a \rightarrow Q \\
 &\quad P_T \sqsubseteq_{TF} \text{atomic}\{Q\} \\
 &\quad P_T \sqsubseteq_{TF} Q \text{ within}[d] \quad \text{for any } d \\
 &\quad P_T \sqsubseteq_{TF} Q \text{ deadline}[d] \quad \text{for any } d
 \end{aligned}$$

So, *PSP* is a trace timewise refinement of *UPSP* which means that PSP_{PE} is a trace timewise refinement of PIP_{PE} :

$$UPSP_T \sqsubseteq_{TF} PSP \Rightarrow PIP_{PE} =_T UPSP_{PE} \sqsubseteq_{TF} PSP_{PE}$$

.

10.3 Case Study: Schedulability Analysis of R2-G2Px Robot Control Software

In this section, we present a case study of our schedulability analysis approach. The subject of the case study is the CSP model of a robot control software. We assume there is an existing PIP of the control software and then we construct and analyse the corresponding PSP.

The case study is based on the two-wheeled robot described in Chapter 3. First, we explain the behaviour requirements of the control software for the robot. Second, we

present the PIP of the control software which is modelled as an untimed process according to the specifications of the software. Third, we present the construction of the PSP. Last, we analyse schedulability of the PSP which reveals that the PSP is not schedulable. Then we show how to locate the cause of unschedulability and make the PSP schedulable by modifying the hardware mapping of the tasks.

10.3.1 Robot Behaviour Specification

As explained in Chapter 3, the R2-G2Px robot is a mobile, two-wheeled differential drive robot. It features many sensors and actuators therefore it is possible to define a behaviour with multiple timing requirements for reading sensors and writing to actuators. For our case study, we assume that it is equipped with a dual-core CPU.

The robot is required to drive forward while following a black line printed on the floor. It is on the line if at least one of the line sensors detects the line. Additionally, the robot should keep a predefined distance to the obstacles that it detects in front of it. As a safety requirement, the robot needs to stop when it gets out of line (whenever neither of the line sensors detects the line) or it bumps into an obstacle. An initial control design results in a two-layer cascaded design composed of a sequence controller and a loop controller. The timing requirement for the initial control design to work is that the sequence and loop controllers have periods of 80 and 20 milliseconds, respectively.

10.3.2 Platform-Independent Process (PIP) of Control Software

The PIP of the control software serves as a model of the software architecture that focuses on concurrent components and their synchronization. The model abstracts away from the control algorithms; all the algorithmic computations are represented by the placeholder task events in the PIP. We assume the processes in the PIP that include control computations (denoted by the task events) collaboratively achieve the control goals if the PIP obeys the specifications of the initial control design that defines the correct way for the processes to synchronize with each other and the frequencies they should be executed at.

Top Level Robot_Control Process

The PIP modeling the robot control software is called `ROBOT_CONTROL`. The external events of the `ROBOT_CONTROL` process are shown in Figure 10.4. `ROBOT_CONTROL` is a parallel combination of two sub-processes: `SEQUENCE_CONTROL` and `LOOP_CONTROL`. The process diagram and the PAT description for the top level decomposition of the `ROBOT_CONTROL` are shown in Figure 10.5 and in Listing 10.10.

Denoted by the dashed arrow in Figure 10.5, the parallel processes communicate with each other asynchronously using a shared variable `motor_speed_setpoints`. This is due to the fact that the processes are required to execute at different frequencies and that `LOOP_CONTROL` needs to oversample the latest value provided by `SEQUENCE_CONTROL`. The reading and writing to the shared variable are represented by internal events in the description of `LOOP_CONTROL` and `SEQUENCE_CONTROL`.

Figure 10.4: External events of `ROBOT_CONTROL` process.

Figure 10.5: Process diagram showing the top level composition of the `ROBOT_CONTROL` process.

Sequence_Control Process

The process `SEQUENCE_CONTROL` models the sequence controller component of the control software. The PAT description of `SEQUENCE_CONTROL` is shown in Listing 10.11. As illustrated in Figure 10.6, at the top level it is composed of three parallel components which communicate with each other through synchronous channels.

`OBJECT_DISTANCE` component reads distance measurements from distance sensors and computes a single value for the distance of the robot to the obstacle. The distance computation task is denoted the event `compute_object_distance` in the process description.

`ROBOT_SPEED` implements a distance controller that periodically checks the distance of the robot to the obstacle and computes a set point value for the speed. The speed computation task is denoted by the event `compute_robot_speed` in the process description.

`MOTOR_SPEED` component computes speed set point values for the two motors of the robot after reading the robot speed set point and the line sensor values. If the

```
1 ROBOT_CONTROL = SEQUENCE_CONTROL || LOOP_CONTROL;
```

Listing 10.10: Description of `ROBOT_CONTROL` process.

```

1 OBJECT_DISTANCE = read_distance_sensors ->
2     compute_object_distance -> distance_meas -> Skip;
3
4 ROBOT_SPEED = distance_meas -> compute_robot_speed -> robot_speed -> Skip;
5
6 MOTOR_SPEED = robot_speed -> read_line_sensors -> Skip;
7     (offline -> set_zero_speed_offline -> Skip
8     []
9     ontheline ->
10        ((compute_motor_speed_setpoint.0 -> Skip)
11         ||| (compute_motor_speed_setpoint.1 -> Skip)));
12     write_motor_speed_setpoints -> Skip;
13
14 SEQUENCE_CONTROL = (OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED);
15     SEQUENCE_CONTROL;

```

Listing 10.11: Description of `SEQUENCE_CONTROL` process.

line sensor values indicate that the robot is off the line (denoted by the first external choice operand starting with `offline` in Listing 10.11) then motor speed set points are set to zero; no set point computation is required. Otherwise the robot is still on the line and individual motor speed setpoints are computed with the aim of keeping the robot on the line. The computational tasks are denoted by the two `compute_motor_speed.i` events in the process description.

Figure 10.6: Process diagram showing the top level composition of `SEQUENCE_CONTROL` process.

Loop_Control Process

The process `LOOP_CONTROL` models the loop controller component that controls the speed of the motors via pulse-width modulation (PWM). In order to compute the PWM values, it reads the speed setpoints for the motors, which are output by `SEQUENCE_CONTROL`, and the odometer values, which are used to compute the actual speed of the wheels. Description of the `LOOP_CONTROL` is shown in Listing 10.12.

As illustrated in Figure 10.7, at the top level, `LOOP_CONTROL` is the parallel combination of three processes. The process `SPEEDOMETERS` reads wheel positions from odometers and computes the actual turning speeds of the wheels; the speed computation is denoted by the `compute_actual_speed.m_id` events.

`SAFETY_FILTER` reads the bump sensor and the speed setpoint values for the the

Figure 10.7: Process diagram of `LOOP_CONTROL` process.

```

1 SPEEDOMETER(m_id) = compute_actual_speed.m_id -> speed_meas.m_id -> Skip;
2 SPEEDOMETERS = read_odometers -> (SPEEDOMETER(0) ||| SPEEDOMETER(1));
3
4 SAFETY_FILTER = read_motor_speed_setpoints -> read_bump_sensor ->
5   (safe_speed.0 -> Skip ||| safe_speed.1 -> Skip);
6
7 MOTOR_CONTROL(m_id) = (safe_speed.m_id -> Skip ||| speed_meas.m_id -> Skip);
8   compute_motor_control.m_id -> Skip;
9
10 MOTOR_CONTROLS = (MOTOR_CONTROL(0) ||| MOTOR_CONTROL(1)); write_pwm -> Skip;
11
12 LOOP_CONTROL = (SPEEDOMETERS || SAFETY_FILTER || MOTOR_CONTROLS);
13   LOOP_CONTROL;

```

Listing 10.12: Description of `LOOP_CONTROL` process.

motors. Unless there is a bump, it passes the setpoint values to `MOTOR_CONTROL` processes unaltered. In case of a bump, it feeds zero speed values to the `MOTOR_CONTROL` processes.

`MOTOR_CONTROLS` process models the loop controllers that control the speed of the motors. The speed set points are provided by `SAFETY_FILTER` component and the measured wheel speeds are provided by `SPEEDOMETERS`. The control computations are denoted by `compute_motor_control.m_id` events.

10.3.3 Construction of the Platform-Specific Process (PSP)

We assume that the presented PIP `ROBOT_CONTROL` satisfies all the safety specifications of the system. In this section we explain how we construct a PSP from it.

The construction starts with refining `ROBOT_CONTROL` into a timed process with platform-independent timing. Since we assume that the only events that take time are

```

1 #define _SEQ_PERIOD 80;
2 SEQUENCE_CONTROL = ((OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED)
3                     || Wait[_SEQ_PERIOD]);
4 SEQUENCE_CONTROL;
5
6 #define _LOOP_PERIOD 20;
7 LOOP_CONTROL = ((SPEEDOMETERS || SAFETY_FILTER || MOTOR_CONTROLS)
8                || Wait[_LOOP_PERIOD]);
9 LOOP_CONTROL;

```

Listing 10.13: Descriptions of `SEQUENCE_CONTROL` and `LOOP_CONTROL` processes after they are instrumented with periodic delays.

the task events, we convert all the other non-placeholder events in `ROBOT_CONTROL` to urgent events. Then, we modify `SEQUENCE_CONTROL` and `LOOP_CONTROL` processes to have periods of 80 and 20 time units (one time unit signifies one millisecond) by instrumenting them with `Wait` processes. `SEQUENCE_CONTROL` and `LOOP_CONTROL` are shown in Listing 10.13 after these modifications. We omit here the rest of description of the `ROBOT_CONTROL` process obtained after these modifications.

In the next step, we specify the best and worst case execution times, priorities and mapping of the tasks in `ROBOT_CONTROL`. Each task is assigned with a unique id and the task and mapped process arrays are specified (Listing 10.14).

When the sum (28 time units) of the WCET's of the tasks ($t_{id=0,1,2,3}$) in a single iteration of `LOOP_CONTROL` is inspected, it can be seen that it is larger than the expected period (20 time units) of `LOOP_CONTROL`. Therefore, in order to keep the execution of a single iteration of `LOOP_CONTROL` within 20 time units, its tasks are distributed into two CPU cores (Listing 10.14). Also, due to the shorter period of `LOOP_CONTROL`, priorities of the tasks in `LOOP_CONTROL` are assigned with a higher priority value than that of the tasks in `SEQUENCE_CONTROL` (Listing 10.14).

As the last step of constructing the PSP, we replace all the placeholder task events in `ROBOT_CONTROL` with `TASK` processes. The `task_arr` index (Listing 10.14) of each replaced task event is given as the argument to the `TASK` process replacing it. We also define `_NUM_OF_CPUS` as 2 and combine `ROBOT_CONTROL` with `CPUS` process, which is the combination of two CPU processes in this case. We omit here the description of the resulting PSP.

10.3.4 Analysis of the Platform-Specific Process (PSP)

There are two time constrained processes in `ROBOT_CONTROL`; deadlines defined for single iterations of the `SEQUENCE_CONTROL` and `LOOP_CONTROL` processes are 80 and 20 time units, respectively. We mark start and end points of time constrained processes and specify the deadline array as shown in Listing 10.15. We also combine the PSP constructed from `ROBOT_CONTROL` with the `DEADLINES` process. Then, we


```

1 //task_arr[t_id][attr]
2 //attr: 0 -> BCET
3 //      1 -> WCET
4 //      2 -> MP_ID
5 var task_arr[8][3] = [4,7,0, //t_id=0: compute_actual_speed.0
6                        4,7,1, //      1: compute_actual_speed.1
7                        5,7,2, //      2: compute_motor_control.0
8                        5,7,3, //      3: compute_motor_control.1
9                        3,7,4, //      4: compute_object_distance
10                       1,3,5, //      5: compute_robot_speed
11                       2,3,5, //      6: compute_motor_speed_setpoint.0
12                       2,3,5]; //      7: compute_motor_speed_setpoint.1
13
14 //mp_arr[mp_id][attr]
15 //attr: 0 -> PRIORITY
16 //      1 -> CPU_ID
17 var mp_arr[6][2] = [2,0, //mp_id=0: SPEEDOMETER(0)
18                    2,1, //      1: SPEEDOMETER(1)
19                    2,1, //      2: MOTOR_CONTROL(0)
20                    2,0, //      3: MOTOR_CONTROL(1)
21                    1,1, //      4: OBJECT_DISTANCE
22                    1,0]; //      5: ROBOT_SPEED || MOTOR_SPEED

```

Listing 10.14: Task and mapped process attribute arrays specified for ROBOT_CONTROL. Schedulability analysis of ROBOT_CONTROL with this configuration fails to hold.

formulate two assertions: the schedulability assertion and the assertion for deadlock freedom (Listing 10.15).

We perform dense-time model checking on the schedulability assertion using PAT's verification engine with "Shortest Witness Trace with Zone Abstraction" setting. On a notebook PC with Intel Core Duo 2.53GHz CPU and 4GB of memory, the verification fails in 3 seconds giving a shortest witness trace.

Inspection of the witness trace reveals that when the tasks with ids 0 and 1, which correspond to `compute_actual_speed.m_id` tasks in `LOOP_CONTROL`, are dispatched, the execution of the task 0 may finish earlier and, since the next highest priority task (`t_id=3`) in the same CPU (CPU 1) waits to be released after the task 1 is finished, the task 4, a lower priority task, is scheduled in CPU 1. In this case, the task with id 3 is dispatched after the task 4 and, if they both take their WCET's to finish, the `LOOP_CONTROL` process misses its deadline.

This is a typical example of a multiprocessor scheduling anomaly involving a counter-intuitive, hard-to-catch timing behaviour. A locally faster execution (the task with id 0 finishes before its WCET) leads to an increase of the overall execution time (execution time for a single iteration of `LOOP_CONTROL` process).

In order to fix this, we can modify the mapped process array by swapping the `CPU_ID`s of mapped processes with id's 2 and 3 as shown in Listing 10.16. Verification of the schedulability assertion with this modified mapped-process array ends in 16 seconds

```

1 #define _DEADLINE_COUNT 2;
2 var deadline_arr[_DEADLINE_COUNT]= [_SEQ_PERIOD,
3                                     _LOOP_PERIOD];
4
5 SEQUENCE_CONTROL = ((d_start.0 ->> (OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED);
6                     d_end.0 ->> Skip)
7                     ||| Wait[_SEQ_PERIOD]);
8 SEQUENCE_CONTROL;
9
10 LOOP_CONTROL = ((d_start.1 ->> (SPEEDOMETERS || SAFETY_FILTER || MOTOR_CONTROLS);
11                 d_end.1 ->> Skip)
12                 ||| Wait[_LOOP_PERIOD]);
13 LOOP_CONTROL;
14
15 PSP_ROBOT_CONTROL = SEQUENCE_CONTROL || LOOP_CONTROL || CPUS || DEADLINES;
16
17 #assert PSP_ROBOT_CONTROL != []!(missed.0 || missed.1);
18 #assert PSP_ROBOT_CONTROL deadlockfree;

```

Listing 10.15: Description of `PSP_ROBOT_CONTROL` after it is instrumented to specify two time constrained processes in `SEQUENCE_CONTROL` and `LOOP_CONTROL`. `PSP_ROBOT_CONTROL` is combined with `DEADLINES` process. The schedulability assertion and the assertion for deadlock-freedom are formulated.

indicating that the assertion is valid. The assertion for deadlock freedom is also verified to hold for this modified configuration in 13 seconds.

10.4 Conclusions

In this chapter, we described how a vertical contract can be formed by taking conjunction of a control design contract and a platform contract so that the resulting vertical contract reflects the schedulability of the application expressed by the design contract on the execution platform described by the platform contract. Then we presented a schedulability analysis framework based timed CSP. This framework can be used to check consistency of a vertical contract that is formulated in the described way. How-

```

1 //mp_arr[mp_id][attr]
2 //attr: 0 -> PRIORITY
3 //      1 -> CPU_ID
4 var mp_arr[6][2] = [2,0, //mp_id=0: SPEEDOMETER(0)
5                    2,1, //      1: SPEEDOMETER(1)
6                    2,0, //      2: MOTOR_CONTROL(0)
7                    2,1, //      3: MOTOR_CONTROL(1)
8                    1,1, //      4: OBJECT_DISTANCE
9                    1,0]; //      5: ROBOT_SPEED || MOTOR_SPEED

```

Listing 10.16: The modified mapped process attribute array which makes `PSP_ROBOT_CONTROL` schedulable.

ever, the schedulability analysis framework is general so it can be employed in development processes not using contracts as well.

The schedulability analysis framework is based on dense-time model checking in PAT tool to check the schedulability of control systems modelled in CSP. Analysis is based on dense-time model checking on timed CSP models. The supported scheduling scheme is non-preemptive fixed-priority multi-processor scheduling. Best and worst case execution times of the tasks are incorporated. Our framework enables analysing multiprocessor schedulability of the systems modelled in CSP involving complex task synchronizations and variable task execution times which traditional schedulability analysis methods fall short on solving. Since our framework is based on model checking and the CSP model of the control system is incorporated in the analysis, the results are non-pessimistic.

We also presented a schedulability workflow associated with the proposed schedulability framework. The workflow defines the successive steps to construct and analyse a timed and scheduled, platform-dependent CSP model from an untimed and unscheduled, platform-independent model of the control system. On a control software design for a mobile robot, we demonstrated the proposed schedulability workflow where we constructed and analysed a timed and scheduled CSP model of the control system from an initial untimed and unscheduled CSP model. Schedulability analysis performed on the constructed model revealed that the system is unschedulable due to a multiprocessor scheduling anomaly. To resolve this, we modified the hardware mapping and we verified that the system is schedulable with the new configuration.

A limitation of the proposed schedulability framework is that the communication and context switching actions are assumed to take insignificant amounts of time. This is a valid assumption for the cases where the total execution time is dominated by the computational task executions and the communication and context-switching times are bounded. For such cases the times for the non-computational tasks can be accommodated within the execution times of the computational tasks.

The scalability of the proposed approach is highly dependent on the performance of the verification engine of the PAT tool and the size of state space of the analysed CSP model. Our case study indicates that the schedulability analysis of a small sized system can be performed in less than 20 seconds on a notebook PC of moderate performance. However, the scalability needs to be investigated further in the future to assess the applicability of the proposed method to more complex systems.

The schedulability framework can be extended to support scheduling schemes with dynamic priorities and/or preemption at integer time points. It will be also beneficial to include the communication times in the framework to better support the distributed control systems in which the communication events might take significant amounts of time. However, we note that these extensions would presumably increase the state space of the schedulability problems and decrease the scalability of the proposed method.

Part IV

Reflections

11

Conclusions

In this chapter we summarize our contributions, discuss the current status of the presented work and elaborate on future directions. In Section 1.1, we stated the list of properties of system specification formalisms and methods to get good quality specifications as follows:

- Exact but abstract specifications
- Executable/analysable specifications, not just mathematical descriptions
- Support for hybrid behaviour
- Explicit assumptions on environment
- Explicit quality/validity criteria, such as completeness and consistency, and methods to check them
- Support for component-based descriptions
- Use of appropriate architectures

We also stated a list of desired properties for a system design process following a system specification as follows:

- Seamless integration of specification and design through refinement relations
- Support for an integrated component-based and layered design/development: decomposition and abstraction, and development by multiple parties
- Support for expressing and analysing behaviours of various nature and multiple viewpoints

Based on the stated desired properties of system specification and design processes, the two main goals of this thesis are to utilize an integration of MBD and CBD methods to:

1. Provide a system specification process that describes how to form and validate an embedded control system specification as an A/G contract
2. Describe how a system specification expressed as an A/G contract can be refined by an A/G design contract via controller design and how such a design contract can be analysed

11.1 Results

In Part II of this thesis we addressed the first goal. In this part, we provided a CBD-oriented system-specification process for embedded systems based on a mode-based architecture. For the contracts used in the specification process we provided concrete formalisms, validity criteria, practical methods to perform contract operations and to assess validity of contracts.

In Part III of this thesis we addressed the second goal. In this part, we provided the description of how a specification contract can be refined by a design contract. We described how an observer-based formulation of a specification contract can be used to drive a conventional control design which constitutes the design contract. We described the use of vertical contracts to obtain a contract from the design contract from the schedulability point of view. To analyse the schedulability, we provided a generic schedulability analysis framework based on model checking using timed CSP.

The most important outcome of this work is that we provided, in a contract-based development process, the ability to use different formalisms to express and work with contracts at different parts of the development. We used:

- hybrid contracts and hybrid system simulation for specifying and validating mode behaviour
- timed contracts and model checking for specifying and validating the switching behaviour
- the combination of hybrid and timed contracts, and simulation and model checking in overall system specification and validation
- model checking using timed CSP for analysis of schedulability of a design contract

All in all, we demonstrated the usefulness of combining CBD and MBD.

It is crucial for a development process to be feasible to be employed in practice. As a foundation for this aim, in Chapter 5, we defined an exact but practical formalism for expressing and validating control system specifications with A/G contracts. We defined a concrete formalism for hybrid assertions based on previous works on hybrid automata. Our contribution in this part is that, in accordance with our aims, we employed a hybrid automata formalism in the context of A/G contracts with a focus on defining practically testable observers.

Even with this foundation, the overall development process can be cumbersome since developing large systems involves complex tasks of defining and manipulating many contracts. As we pointed out, checking validity of contracts with hybrid assertions is not feasible or even possible in the case of general hybrid behaviour, to be fully automated. Manual work is needed there, however management and combining the results of manual steps can be easily automated much like in automated theorem proving.

Even though we advocated the necessity of expressing hybrid behaviour in contracts for system specification and, hence, the use of testable observers and hybrid system simulation tools for the sake of practicality, we reflected, in Chapter 7 and in Chapter 8,

that formal verification should be employed wherever it is feasible. Formal verification can be feasible when the specification of a component can be expressed in terms of non-hybrid or timed assertions. The crucial part is to be able to validate a specification with components with hybrid, timed or non-hybrid behaviour. In this direction, our contribution is that, in Chapter 8, we used timed automata formal verification (Uppaal) and hybrid system simulation (20-sim) together to validate contracts with timed and hybrid parts.

Apart from the tools and methods available for analysing behaviour of a component specification, it is as crucial to have the validity criteria to drive the analysis efforts. In this direction, we have defined validity criteria for each component of our mode-based specification architecture. First, we defined formalism-independent criteria in Chapter 4, and then, refined these criteria in Chapter 5 with validity criteria specific for hybrid contracts. In Chapters 6, 7 and 8, we have seen yet the need to refine the validity criteria and the validation methods. We provided specific validity criteria and validation methods for mode-composition contract, mode-switching contract and the overall system contract.

Once the specification of a system is defined and validated, the design process starts aiming to realize the specification. In Part III of our work, given a system specification contract, we provided the means to realize the specification via digital control design and analyse the schedulability aspect of the design on a specific execution platform. Once again, to ensure the process feasible to be employed in practice, we aimed to provide a design process that embraces the conventional control design methods and tools in use today.

In Chapter 9, we sketched how a system specification contract can be refined by a system design contract consisting of a digital control design model which implements the system specification. The digital control design and the validation of the implementation relation should be assisted by a hybrid system simulation tool. In such tools that are used to perform control design, the semantics of a digital control model is based on a form of synchronous block diagram (SBD). Based on SBD semantics, in Chapter 9 we described how one can use the notion of vertical contracts to tie a digital control design model to an implementation of it on a particular execution platform. This opened the way of obtaining a mapping of digital control design implementation to an execution platform.

In Chapter 10, we presented a schedulability analysis framework based on model checking on timed-CSP models. We described, how one can analyse schedulability of the tasks in a digital control implementation on a particular execution platform. Our schedulability framework supports non-preemptive fixed-priority multiprocessor scheduling. However, our contribution regarding the schedulability analysis framework is independent of the context we used it in, that is schedulability analysis for a contract-based control design. It can be used to analyse schedulability of systems modelled in CSP involving complex task synchronizations and variable task execution times. We demonstrated the generic use of our framework on an example of robot control software.

11.2 Open Issues and Future Work

In the A/G contract algebra we described in Subsection 2.2.2, a contract specifies a component in terms of the sets of allowed behaviour for the legal environments and the component. However, a contract cannot ensure that a component or its environment show a particular behaviour. This is indicated by that the refinement and implementation relations in the contract algebra (Subsection 2.2.2) are defined in terms of subset relation. This results in that, after a refinement or an implementation for a given A/G contract with non-empty set of behavior, the intersection of the sets of behaviour of the legal environments and of components might be empty (Subsection 2.2.3). Such a refinement or an implementation would be useless as the component and its environment are not consistent any more.

We identify this as an issue with the A/G contract framework for which, in our work, we provided the following mitigations. With respect to formulating and refining specification contracts during system specification process, we required in Subsection 5.5.3, that, as a part of the consistency criteria for an A/G contract, the implementations and legal environments defined by a contract are consistent with each other. With respect to refining a specification contract with a design contract, our approach of employing conventional control design described in Chapter 9, is immune to this problem as, in a hybrid-system simulation tool, we model and simulate component and environment models together which prevents any inconsistencies.

However, this issue still persists for parts of a contract-based design process that were not scoped in our work. It is a problem in particular when initially consistent contracts are implemented independently by different parties. The resulting implementations might have sets of behaviour with an empty intersection. We believe that this issue cannot be addressed in general, i.e. independent of the particular use of contracts, within the context of the contract framework. An effort to have a contract dictate the associated components to show particular behaviour would invalidate the operations defined in the current contract algebra, for instance the refinement and implementation relations. We believe a feasible solution should be in the form of methodological compensation specific to the particular use of contracts, similar to what we applied in the context of specification contracts. For instance, regarding independent implementation of two consistent contracts, one might be able prevent inconsistency of the implementations by ensuring that the implementations are deadlock free and input-enabled. This direction deserves more investigation.

In the specification workflow (Figure 4.8) we presented, the initial system requirements are defined over the system's external ports and variables and independent of any mode-based requirements decomposition. System requirements would span multiple modes and mode-switching logic in any mode-based decomposition. After the mode-based requirements decomposition and contract-based formalization and validation, we obtain the overall system contract by composing mode contracts and the mode-switching contract. At this stage, the correct decomposition of initial system requirements and correct formalization of requirements in terms of contracts ensure that the overall system contract refines the initial system requirements. An improve-

ment in validating the overall system specification further would be to validate overall system contract against initial system requirements. This would require a sort of formalization of the initial system requirements independent of the mode-based decomposition. Then these requirements can be checked on the overall system contract with the help of hybrid system tools, 20-sim and Uppaal in our case.

Furthermore, after a system specification contract is produced and validated, if there is a need for change or addition, this might require a re-validation of the whole contract, i.e. the repetition of the validation steps in Figure 4.8. An improvement in this area would be to define categories of changes such as addition of a new mode or a change of a mode, and formulate strategies to eliminate unnecessary validation steps and/or simplify involved validation steps.

In Section 4.1, we advocated that a system specification should be solely defined over the environment variables where the problem lies within. According to this view a system specification should be a black-box specification not prematurely dictating any design directions. However, as we stated in Section 9.1, given a system specification contract defined with testable observers, eliminating the internal ports resulting from the mode-based specification architecture introduces non-determinism which makes the specification contract not testable by simulation any more. In order to mitigate this, we suggested to incorporate the internal ports in the design process. As long as the design architecture can be mapped to the specification architecture, to generate internal port values of the specification contract during simulation, this approach is valid. Unfortunately, with this approach the design becomes partly tied to the specification architecture and extra effort is needed to calculate the values of the internal specification ports in the design models.

A future direction in addressing this might involve automated elimination of internal ports from system contract where possible. The aim is to devise a method which, for a given testable hybrid observer and an input variable, will result in a testable observer such its behaviours will match the behaviours of the original observer projected onto the alphabet without the indicated variable. Automated elimination might only be possible for the variables that are exclusively used in discrete transition decisions, i.e. in location invariants and guards for discrete transitions, that do not affect continuous flow. An example is a variable that is only used in an inequality in a transition guard towards a bad state. If a variable is used in a way to affect continuous flow, analysis will resemble hybrid system reachability analysis, which is not feasible.

Regarding contract-based design of hybrid systems, a prominent future research direction is to employ verification by simulation methods (Girard and Pappas, 2006; Alur et al., 2008). Hybrid observers enable employing hybrid system simulation tools to perform checks and validations on hybrid contracts. However, simulation alone suffers from the difficulty to sufficiently cover the input space with individual simulation runs and the inability to cope with non-determinism. Verification by simulation can assist in this area to increase effectiveness of checking whether a hybrid system model implements a given contract or whether a contract refines another contract. A particularly interesting related work is (Duggirala et al., 2013), where authors achieve verifi-

cation of bounded time safety of non-linear dynamical systems with multiple control modes in Simulink/Stateflow. This is done with the help of user provided annotations for control modes that state trajectory convergence and divergence. Within the context of our work, such annotations can easily be provided as extra guarantee assertions in a contract.

A

Frequently Used Symbols and Notations

Sets

$\forall$	for all
$\exists$	there exists
$D(p)$	the domain of values for the variable p
D^*	set of all finite words over the set D
D^ω	set of all infinite words over the set D
$\mathbf{S}$	set of all valuations for a set of variables S
$f : A \rightarrow B$	f maps the domain A to codomain B
$P(X) = 2^X$	the power set of the set X
$A \times B$	cartesian product of two sets

Contract Algebra

$M_1 \times M_2$	composition of two components
$E \models^E C$	E is an environment for C
$M \models^M C$	M is an implementation for C
$C_1 \leq C_2$	C_1 refines C_2
$C_1 \wedge C_2$	conjunction of two contracts
$C_1 \otimes C_2$	composition of two contracts
$C_1 \preceq_c C_2$	C_1 conforms to C_2
$C_1 \preceq_s C_2$	C_1 strongly refines C_2

Hybrid Assertions and Hybrid Automata

$\tau = \{I_i\}_{i=0}^N$	a hybrid time set (Definition 5.1.1)
$\mathbf{\tau}$	the set of all hybrid time sets
$\langle \tau \rangle$	the discrete extent of the hybrid time set τ
$\ \tau\ $	the continuous extent of the hybrid time set τ
$comb_s(\tau, \hat{\tau})$	combination of two hybrid time sets with last interval of τ and first of $\hat{\tau}$ joined (Definition 5.1.3)
$comb_p(\tau, \hat{\tau})$	combination of two hybrid time sets with last interval of τ and first of $\hat{\tau}$ separate (Definition 5.1.3)
$\tau \sqsubseteq \hat{\tau}$	τ is a prefix of $\hat{\tau}$
$\tau \sqsubset \hat{\tau}$	τ is a strict prefix of $\hat{\tau}$
$\tau \sqsupseteq \hat{\tau}$	τ is a postfix of $\hat{\tau}$
$\tau \sqsupset \hat{\tau}$	τ is a strict postfix of $\hat{\tau}$
(τ, w)	a hybrid trajectory (Definition 5.1.4)
$comb_s((\tau, w), (\hat{\tau}, \hat{w}))$	combination of two hybrid time sets with last interval of (τ, w) and first of $(\hat{\tau}, \hat{w})$ joined (Definition 5.1.5)
$comb_p((\tau, w), (\hat{\tau}, \hat{w}))$	combination of two hybrid time sets with last interval of (τ, w) and first of $(\hat{\tau}, \hat{w})$ separate (Definition 5.1.5)
$(\tau, w) \sqsubseteq (\hat{\tau}, \hat{w})$	(τ, w) is a prefix of $(\hat{\tau}, \hat{w})$
$(\tau, w) \sqsubset (\hat{\tau}, \hat{w})$	(τ, w) is a strict prefix of $(\hat{\tau}, \hat{w})$
$(\tau, w) \sqsupseteq (\hat{\tau}, \hat{w})$	(τ, w) is a postfix of $(\hat{\tau}, \hat{w})$
$(\tau, w) \sqsupset (\hat{\tau}, \hat{w})$	(τ, w) is a strict postfix of $(\hat{\tau}, \hat{w})$
$Traj(W)$	the set of all hybrid trajectories over the set of variables W
$Traj^\infty(W)$	the set of all infinite hybrid trajectories over the set of variables W
$join(h)$	the hybrid trajectory resulting from eliminating all the effectless discrete transitions (Definition 5.1.6)
$pr_W(h)$	projection of a hybrid trajectory h onto a set of variables W (Definition 5.1.7)
$pr_W(L)$	projection of an assertion L onto a set of variables W
$pr_W^{-1}(L)$	inverse projection of an assertion L onto a set of variables W
$Exec(H)$	the set of executions of a hybrid automaton H
$Reach(H)$	the set of states reachable by a hybrid automaton H
$Asrt_P(H)$	the assertion defined over an alphabet P by a hybrid automaton H
$OHA_1 OHA_2$	composition of two hybrid automata (Definition 5.2.8)

Acronyms

CBD contract-based design. 4–9, 15, 197, 198

CSP communicating sequential processes. xviii, 6, 9, 10, 13, 111, 164, 167, 169–173, 175–177, 180, 183, 185, 186, 193, 194, 198, 199

DET deadline execution time. 163, 167, 168

IUT implementation under test. 125–127, 129, 131, 132

LET logical execution time. 163, 167, 168

MBD model-based design. 4–6, 15, 16, 197, 198

PIP platform-independent process. 175–180, 182, 184–187, 190

PSP platform-specific process. 175–177, 179, 180, 182–187, 190, 191

SBD synchronous block diagram. 161, 162, 164, 199

ZET zero execution time. 162, 163, 167, 168

Bibliography

- Alur, R., Courcoubetis, C., and Dill, D. (1990). Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on*, pages 414–425.
- Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.
- Alur, R., Kanade, A., Ramesh, S., and Shashidhar, K. C. (2008). Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models. In *Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT '08*, pages 89–98, New York, NY, USA. ACM.
- Alur, R. and Madhusudan, P. (2004). Decision Problems for Timed Automata: A Survey. In Bernardo, M. and Corradini, F., editors, *Formal Methods for the Design of Real-Time Systems*, number 3185 in Lecture Notes in Computer Science, pages 1–24. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-30080-9_1.
- Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W. (2004). TIMES: a tool for schedulability analysis and code generation of Real-Time systems. In Niebert, P. and Larsen, K. G., editors, *Proceedings of FORMATS'03*, Lecture Notes in Computer Science, pages 60–72. Springer Verlag.
- Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.
- Barnes, F. R. M. and Welch, P. H. (2003). Prioritised dynamic communicating and mobile processes. *IEE Proceedings-Software*, 150(2):121–136.
- Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., and Legay, A. (2010). Statistical Abstraction and Model-Checking of Large Heterogeneous Systems. In Hatchiff, J. and Zucca, E., editors, *Formal Techniques for Distributed Systems: Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, pages 32–46. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI: 10.1007/978-3-642-13464-7_4.
- Bauer, S. S., David, A., Hennicker, R., Larsen, K. G., Legay, A., Nyman, U., and Wąsowski, A. (2012). Moving from Specifications to Contracts in Component-Based Design. In Lara, J. d. and Zisman, A., editors, *Fundamental Approaches to Software Engineering*, number 7212 in Lecture Notes in Computer Science, pages 43–58. Springer Berlin Heidelberg.
- Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on UPPAAL. In Bernardo, M. and Corradini, F., editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag.
- Behrmann, G., David, A., and Larsen, K. G. (2006). A Tutorial on UPPAAL 4.0. In Bernardo, M. and Corradini, F., editors, *International School on Formal Methods for*

- the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag.
- Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., and Sofronis, C. (2008). Multiple Viewpoint Contract-Based Specification and Design. In Boer, F. S. d., Bonsangue, M. M., Graf, S., and Roever, W.-P. d., editors, *Formal Methods for Components and Objects*, number 5382 in *Lecture Notes in Computer Science*, pages 200–225. Springer Berlin Heidelberg.
- Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., and Larsen, K. G. (2012). Contracts for System Design. Research Report, INRIA.
- Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., and Larsen, K. G. (2015). Contracts for Systems Design: Methodology and Application cases. Research Report RR-8760, Inria Rennes Bretagne Atlantique.
- Benveniste, A., Caillaud, B., and Passerone, R. (2009). Multi-Viewpoint State Machines for Rich Component Models. *Model-Based Design of Heterogeneous Embedded Systems*.
- Benvenuti, L., Bresolin, D., Collins, P., Ferrari, A., Geretti, L., and Villa, T. (2014). Assume–guarantee verification of nonlinear hybrid systems with Ariadne. *International Journal of Robust and Nonlinear Control*, 24(4):699–724.
- Benvenuti, L., Ferrari, A., Mangeruca, L., Mazzi, E., Passerone, R., and Sofronis, C. (2008a). A contract-based formalism for the specification of heterogeneous systems. In *Forum on Specification, Verification and Design Languages, 2008. FDL 2008*, pages 142–147.
- Benvenuti, L., Ferrari, A., Mazzi, E., and Vincentelli, A. L. S. (2008b). Contract-Based Design for Computation and Verification of a Closed-Loop Hybrid System. In Egerstedt, M. and Mishra, B., editors, *Hybrid Systems: Computation and Control*, number 4981 in *Lecture Notes in Computer Science*, pages 58–71. Springer Berlin Heidelberg.
- Bezemer, M. M., Wilterdink, R. J., and Broenink, J. F. (2011). Luna: Hard real-time, multi-threaded, csp-capable execution framework. In Welch, P. H., Sampson, A. T., Pedersen, J. B., Kerridge, J. M., Broenink, J. F., and Barnes, F. R., editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam. IOS Press.
- Boström, P. (2011). Contract-Based Verification of Simulink Models. In Qin, S. and Qiu, Z., editors, *Formal Methods and Software Engineering*, number 6991 in *Lecture Notes in Computer Science*, pages 291–306. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-24559-6_21.
- Boström, P., Linjama, M., Morel, L., Siivonen, L., and Waldén, M. (2007). Design and validation of digital controllers for hydraulics systems. In *In The 10th Scandinavian International Conference on Fluid Power*, pages 227–241.

- Brémond-Grégoire, P. and Lee, I. (1997). A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189(1-2):179–219.
- Broenink, J. F. (1999). 20-sim software for hierarchical bond-graph/block-diagram models. *Simulation Practice and Theory*, 7(5):481 – 492.
- Broenink, J. F., Bakkers, A. W. P., and Hilderink, G. H. (1999). Communicating Threads for Java. In Cook, B. M., editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262.
- Brown, N. C. C. (2007). CCSP2: A Many-to-Many Threading Model for Multicore Architectures. In McEwan, A. A., Ifill, W., and Welch, P. H., editors, *Communicating Process Architectures 2007*, pages 183–205.
- Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., and Ratiu, D. (2010). Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE*, 98(4):526–545.
- Burns, A. (1994). Preemptive priority-based scheduling: an appropriate engineering approach. In Son, S. H., editor, *Advances in real-time systems*, pages 225–248. Prentice Hall, Upper Saddle River, NJ, USA.
- Buttazzo, G. C. (2004). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2nd edition.
- Carlioni, L. P., Passerone, R., Pinto, A., and Angiovanni-Vincentelli, A. L. (2006). Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2):1–193.
- Controllab Products (2017). 20-sim.
- Dassault Systemes (2017). DYMOLA - Dynamic Modeling Labrotary.
- David, A., Illum, J., Larsen, K. G., and Skou, A. (2010). Model-based framework for schedulability analysis using uppaal 4.1. In *Model-Based Design for Embedded Systems*, pages 93–119. CRC Press.
- Derler, P., Lee, E. A., Tripakis, S., and Törngren, M. (2013). Cyber-physical system design contracts. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems, ICCPS '13*, pages 109–118, New York, NY, USA. ACM.
- Duggirala, P. S., Mitra, S., and Viswanathan, M. (2013). Verification of Annotated Models from Executions. In *Proceedings of the Eleventh ACM International Conference on Embedded Software, EMSOFT '13*, pages 26:1–26:10, Piscataway, NJ, USA. IEEE Press.
- Esterel Technologies (2016). SCADE Suite.
- Faulk, S., Brackett, J., Ward, P., and Kirby, J., J. (1992). The Core method for real-time requirements. *IEEE Software*, 9(5):22–33.
- Feiler, P., Gluch, D., and Hudak, J. (2006). The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute, Carnegie Mellon University.
- Fersman, E., Krcal, P., Pettersson, P., and Yi, W. (2007). Task automata: Schedulability,

- decidability and undecidability. *Information and Computation*, 205(8):1149–1172.
- Frehse, G. (2008). PHAVer: algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer*, 10(3):263–279.
- Frehse, G. (2015). An Introduction to Hybrid Automata, Numerical Simulation and Reachability Analysis. In Drechsler, R. and Kühne, U., editors, *Formal Modeling and Verification of Cyber-Physical Systems*, pages 50–81. Springer Fachmedien Wiesbaden. DOI: 10.1007/978-3-658-09994-7_3.
- Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., and Maler, O. (2011). SpaceEx: Scalable Verification of Hybrid Systems. In Ganesh Gopalakrishnan, S. Q., editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer.
- Friedenthal, S., Moore, A., and Steiner, R. (2009). OMG SysML Tutorial.
- Fritzson, P. (2003). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Press.
- Gibson-Robinson, T., Armstrong, P., Boulgakov, A., and Roscoe, A. W. (2014). FDR3 — A Modern Refinement Checker for CSP. In Ábrahám, E. and Havelund, K., editors, *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 187–201. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI: 10.1007/978-3-642-54862-8_13.
- Girard, A. and Pappas, G. J. (2006). Verification Using Simulation. In *Proceedings of the 9th International Conference on Hybrid Systems: Computation and Control, HSCC'06*, pages 272–286, Berlin, Heidelberg. Springer-Verlag.
- Gunter, C., Gunter, E., Jackson, M., and Zave, P. (2000). A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43.
- Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M. (1990). STATEMATE a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.
- Haskins, C. (2012). *Incose Systems Engineering Handbook*. Incose, San Diego, California.
- Henzinger, T. (1996). The theory of hybrid automata. In *Eleventh Annual IEEE Symposium on Logic in Computer Science, 1996. LICS '96. Proceedings*, pages 278–292.
- Hessel, A., Larsen, K. G., Mikucionis, M., Nielsen, B., Pettersson, P., and Skou, A. (2008). Testing Real-Time Systems Using UPPAAL. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal Methods and Testing*, number 4949 in Lecture Notes in Computer Science, pages 77–117. Springer Berlin Heidelberg.
- Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.
- Jacklin, S. A., Lowry, M. R., Schumann, J., Gupta, P. P., Bosworth, J. T., Zavala, E., Kelly,

- J. W., Hayhurst, K., Belcastro, C. M., and Belcastro, C. M. (2004). Verification Validation and Certification Challenges for Adaptive Flight-Critical Control System Software. *Collection of Technical Papers - AIAA Guidance, Navigation, and Control Conference*, 4.
- Jackson, M. (2005). Problem frames and software engineering. *Information and Software Technology*, 47(14):903–912.
- Jaffe, M. S. (2011). Minimal embedded robustness requirements analysis. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 7E2–1–7E2–10.
- Jahanian, F. and Mok, A. K. (1994). Modechart: A Specification Language for Real-Time Systems. *IEEE Trans. Softw. Eng.*, 20(12):933–947.
- Korsgaard, M. and Hendseth, S. (2008). Combining EDF scheduling with occam using the toc programming language. In Welch, P. H., Stepney, S., Polack, F., Barnes, F. R. M., McEwan, A. A., Stiles, G. S., Broenink, J. E., and Sampson, A. T., editors, *The thirty-first Communicating Process Architectures Conference, CPA 2008, organised under the auspices of WoTUG and the Department of Computer Science of the University of York, York, Yorkshire, UK, 7-10 September 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 55–66. IOS Press.
- Korsgaard, M. and Hendseth, S. (2011). Schedulability analysis of malleable tasks with arbitrary parallel structure. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, volume 1, pages 3–14. IEEE.
- Krcal, P., Stigge, M., and Yi, W. (2007). Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In *Proceedings of the 5th international conference on Formal modeling and analysis of timed systems, FORMATS’07*, pages 274–289, Berlin, Heidelberg. Springer-Verlag.
- Kuijper, W. and van de Pol, J. C. (2009). Compositional Control Synthesis for Partially Observable Systems. In Bravetti, M. and Zavattaro, G., editors, *CONCUR 2009 - Concurrency Theory*, Lecture Notes in Computer Science, pages 431–447. Springer Verlag. DOI: 10.1007/978-3-642-04081-8_29.
- Larsen, K. G., Mikucionis, M., and Nielsen, B. (2004). Online Testing of Real-time Systems Using Uppaal. In Grabowski, J. and Nielsen, B., editors, *Formal Approaches to Software Testing*, number 3395 in Lecture Notes in Computer Science, pages 79–94. Springer Berlin Heidelberg.
- Larsen, K. G., Mikucionis, M., and Nielsen, B. (2009). Uppaal Tron User Manual.
- Larsen, K. G. and Thomsen, B. (1988). A modal process logic. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*, pages 203–210.
- Legay, A., Delahaye, B., and Bensalem, S. (2010). Statistical Model Checking: An Overview. In Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., and Tillmann, N., editors, *Runtime Verification: First International Conference, RV2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 122–135. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI: 10.1007/978-3-642-16612-9_11.

- Leveson, N. (2000). Completeness in Formal Specification Language Design for Process-control Systems. In *Proceedings of the Third Workshop on Formal Methods in Software Practice*, FMSP '00, pages 75–87, New York, NY, USA. ACM.
- Leveson, N., Heimdahl, M., Hildreth, H., and Reese, J. (1994). Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707.
- Leveson, N. G. (2004). Role of software in spacecraft accidents. *Journal of spacecraft and Rockets*, 41(4):564–575.
- Lutz, R. (1993). Analyzing software requirements errors in safety-critical, embedded systems. In , *Proceedings of IEEE International Symposium on Requirements Engineering*, 1993, pages 126–133.
- Lygeros, J., Johansson, K., Sastry, S., and Egerstedt, M. (1999). On the existence of executions of hybrid automata. In *Proceedings of the 38th IEEE Conference on Decision and Control*, 1999, volume 3, pages 2249–2254 vol.3.
- Lygeros, J., Tomlin, C., and Sastry, S. (2008). Hybrid Systems: Modeling, Analysis and Control. Technical report.
- Lynch, N., Segala, R., and Vaandrager, F. (2003). Hybrid I/O automata. *Information and Computation*, 185(1):105–157.
- Maraninchi, F. and Remond, Y. (1998). Mode-automata: About modes and states for reactive systems. In Hankin, C., editor, *Programming Languages and Systems*, number 1381 in Lecture Notes in Computer Science, pages 185–199. Springer Berlin Heidelberg.
- Meyer, B. (1992). Applying "Design by Contract". *Computer*, 25(10):40–51.
- Mikučionis, M., Larsen, K. G., Rasmussen, J. I., Nielsen, B., Skou, A., Palm, S. U., Pedersen, J. S., and Hougaard, P. (2010). Schedulability analysis using uppaal: Herschel-Planck case study. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ISoLA'10, pages 175–190, Berlin, Heidelberg. Springer Verlag.
- Miller, S. P. (1998). Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 44–53, New York, NY, USA. ACM.
- Mousavi, M., Reniers, M., Basten, T., and Chaudron, M. (2004). PARS: A process algebra with resource and schedulers. In Larsen, K. G. and Niebert, P., editors, *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (Formats'03)*, volume 2791 of *Lecture Notes in Computer Science*, pages 134–150. Springer-Verlag, Berlin, Germany.
- Nuzzo, P., Sangiovanni-Vincentelli, A., Bresolin, D., Geretti, L., and Villa, T. (2015). A Platform-Based Design Methodology With Contracts and Related Tools for the Design of Cyber-Physical Systems. *Proceedings of the IEEE*, 103(11):2104–2132.
- Nuzzo, P., Xu, H., Ozay, N., Finn, J., Sangiovanni-Vincentelli, A., Murray, R., Donze, A., and Seshia, S. (2014). A Contract-Based Methodology for Aircraft Electric Power System Design. *IEEE Access*, 2:1–25.

- Object Management Group (OMG) (2017). Unified Modeling Language™ (UML®) Version 2.5.
- Oguz, O., Broenink, J. F., and Mader, A. (2012). Schedulability Analysis of Timed CSP Models Using the PAT Model Checker. In Welch, P. H., Barnes, F. R. M., Chalmers, K., Pedersen, J. B., and Sampson, A. T., editors, *Communicating Process Architectures 2012*, pages 65–88.
- Orlic, B. and Broenink, J. F. (2004). Redesign of the c communicating threads library for embedded control systems. In Karelse, F., editor, *5th Progress Symposium on Embedded Systems, Nieuwegein, The Netherlands*, pages 141–156. STW Technology Foundation.
- Parnas, D. L. (2010). From Requirements to Architecture. In *Proceedings of the 2010 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the 9th SoMeT_10*, pages 3–36, Amsterdam, The Netherlands, The Netherlands. IOS Press.
- Parnas, D. L. and Madey, J. (1995). Functional Documents for Computer Systems. *Sci. Comput. Program.*, 25(1):41–61.
- Paynter, S. (1996). Real-Time Mode-Machines. In *Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT '96*, pages 90–109, London, UK, UK. Springer-Verlag.
- Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., and Passerone, R. (2011). A Modal Interface Theory for Component-based Design. *Fundam. Inf.*, 108(1-2):119–149.
- Raskin, J.-F. (2005). An Introduction to Hybrid Automata. In Hristu-Varsakelis, D. and Levine, W. S., editors, *Handbook of Networked and Embedded Control Systems*, Control Engineering, pages 491–517. Birkhäuser Boston.
- Reed, G. M. (1990). A hierarchy of domains for real-time distributed computing. In Main, M., Melton, A., Mislove, M., and Schmidt, D., editors, *Mathematical Foundations of Programming Semantics: 5th International Conference Tulane University, New Orleans, Louisiana, USA March 29–April 1, 1989 Proceedings*, pages 80–128, New York, NY. Springer New York.
- Reed, G. M. and Roscoe, A. W. (1988). A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261.
- Roscoe, A. W. (1997). *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA.
- Safeware Engineering Corporation (2017). Software Hazard Analysis.
- Sangiovanni-Vincentelli, A. L., Damm, W., and Passerone, R. (2012). Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *Eur. J. Control*, 18(3):217–238.
- Schneider, S. (1999). *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, New York, NY, USA.
- SGS-Thomson Microelectronics (1995). Occam 2.1 reference manual. Technical re-

- port, SGS-THOMSON Microelectronics Limited.
- Sun, J., Liu, Y., Dong, J. S., Liu, Y., Shi, L., and André, E. (2013). Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Trans. Softw. Eng. Methodol.*, 22(1):3:1–3:29.
- Sun, J., Liu, Y., Dong, J. S., and Pang, J. (2009a). PAT: Towards Flexible Verification under Fairness. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer.
- Sun, J., Liu, Y., Dong, J. S., and Zhang, X. (2009b). Verifying stateful timed CSP using implicit clocks and zone abstraction. In Breitman, K. and Cavalcanti, A., editors, *Proceedings of the 11th IEEE International Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600. Springer.
- The Mathworks Inc. (2016a). Simulink - Simulation and Model-Based Design.
- The Mathworks Inc. (2016b). State Machine - Stateflow - Simulink - MathWorks Benelux.
- Thompson, J. M., Whalen, M. W., and Heimdahl, M. P. E. (2000). Requirements Capture and Evaluation in Nimbus: The Light-Control Case Study. *Journal of Universal Computer Science*, 6(7):731–757. [http://www.jucs.org/jucs_6_7/requirements_capture_and_evaluation].
- Tripakis, S. (2008). Synchronous Block Diagrams. Technical report, Cadence Research Laboratories.
- Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Utting, M., Pretschner, A., and Legeard, B. (2012). A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22(5):297–312.
- Welch, P. H. and Barnes, F. R. M. (2005). Communicating mobile processes: introducing occam-pi. In *In 25 Years of CSP*, pages 175–210. Springer Verlag.
- Wieringa, R. J. (1996). *Requirements Engineering: Frameworks for Understanding*. John Wiley & Sons, Inc., New York, NY, USA.
- Zowghi, D. and Gervasi, V. (2003). On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993–1009.