Automated model-based testing of hybrid systems

Automated Model-based Testing of Hybrid Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 10 februari 2009 om 16.00 uur

door

Michiel Pieter Willem Jacob van Osch

geboren te Deurne

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J.C.M. Baeten
en
prof.dr.ir. J.E. Rooda

Copromotor:
dr. S.P. Luttik

The only way to pass the test is to take the test. It is inevitable.
– Elder Regal Black Swan (Aboriginal tribe leader)

IPA Dissertation series 2009-04

Printed by the University Press Facilities, Eindhoven

Back cover image: A waferstepper machine (© Copyright 2006, ASML)

Cover design by Michiel van Osch

The work on this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The author was employed at the Technische Universiteit Eindhoven. This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Osch van, Michiel Pieter Willem Jacob

Automated Model-based Testing of Hybrid Systems /
door Michiel Pieter Willem Jacob van Osch. –
Eindhoven : Technische Universiteit Eindhoven, 2009.
Proefschrift. – ISBN 978-90-386-1524-0.
NUR 993

Preface

In 2001 I graduated from the Technische Universiteit Eindhoven within the Formal Methods Group. My Master's project concerned modelchecking the CAN bus Standard. After my graduation I felt I wanted a more practical background in software engineering. Therefore, I joined the post Master's program on software technology OOTI. In 2003 I graduated from this program after a final project on client side caching of dynamic web pages at TNO Telecom. After this project I started looking for work that would best fit my professional skills, my personal skills, and my personal interests.

At that time the TANGRAM project crossed my path. The TANGRAM project was a four year research project on model-based test and integration methods for complex high-tech systems. The carrying partners where ASML and the embedded systems institute ESI. ASML is the world leading producer of waferstepper machines. The universities involved where the Technische Universiteit Eindhoven (by the Mechanical Engineering department and the Mathematics & Computer Science department), Twente University, Delft University of Technology, and the Radboud Universiteit Nijmegen. The industry partners besides ASML were Science [&] Technology, the National Aerospace Labratory (NLR) and the Netherlands Organization for Applied Scientific Research (TNO). The approach of the TANGRAM project was to use industry as a laboratory. Scientific research should be developed, tested, and applied in industry directly. Within the TANGRAM project the focus lay on: better scheduling of integration and tests, test automation, early integration, creating a uniform (test) infrastructure, and faster diagnoses.

This was exactly what I was looking for. It allowed me to apply and further develop both my background in theoretical computer science and practical software engineering. It provided the opportunity to both work with and be in contact with research at the university and work within an industrial environment. The research presented in this thesis was part of the test automation focus area. The goal of my research was to develop a model-based test theory for hybrid systems and to validate the applicability of this theory in an industrial setting. In this thesis I present the results of this research.

First of all I thank professor Jos Baeten, Luud Engels and Tom Brugman for providing me the opportunity to perform this Ph.D. project within the Formal Methods Group and at ASML. Second, I thank Bas Luttik for his supervision, his many good

advices and his patience in reviewing the many revisions of my work. Third, I thank my TANGRAM team members, especially: Roel Boumen, Niels Braspenning, Henrik Bohnenkamp, Will Denissen, Ivo de Jong, Jurryt Pietersma, Jan Tretmans, Rene de Vries, and Tim Willemse for their useful input to my research and creating a fruitful working atmosphere. Fourth, I thank Frank Stappers who was a very cheerful and active Master's student and a good help in my research as well. Fifth, I thank my fellow F-buckets for their input and a very enjoyable working atmosphere. Sixth, I thank everybody else who gave input for my research, especially: Ramon Schiffelers, Albert Hofkamp, professor Koos Rooda, and the members of my defense committee.

A Ph.D. research is not only a professional journey but also a personal journey. I thank my mother, my father, and my brother for their support throughout my life and I do not forget to thank my friends as well. Five years ago I started this Ph.D. research as a bachelor. Now, I have a loving wife and at the time of writing this preface, a one week old son. Last, but it should be first, I thank my wife Barbara for her endless patience with me working on this thesis and for her emotional support. I thank my son Daniel for bringing such joy in my life. I dedicate this thesis to them:

to Barbara and Daniel.

Contents

1

Introduction

In this thesis we develop a formal theory and prototype tool for automated model-based testing of hybrid systems. With a case study we also show that this method of testing is useful for industry.

In this introduction we first explain which systems we consider to be hybrid systems. Then, we explain how automated model-based testing fits in a set of different testing approaches. In this thesis a formal approach to model-based testing is taken. Therefore we explain what is needed for a formal model-based test theory. Specifically, our theory for hybrid systems is based on three existing input-output conformance test theories. We give a brief overview of these input-output conformance test theories. We conclude this chapter with a detailed overview of the remainder of thesis.

1.1 Hybrid Systems

A hybrid system is a system with both discrete-event behavior and continuous behavior. By discrete-event behavior we mean (instantaneous) actions. E.g. A hybrid system can display discrete-event behavior by software but also by its hardware, e.g. buttons. For a system, by continuous behavior we mean the behavior of physical quantities. A hybrid system may perform continuous behavior by e.g. actuators and observe continuous behavior through e.g. sensors.

A thermostat is an example of a hybrid system with continuous input and discrete-event output. It observes the temperature of a room through a sensor, and it controls a heating system through signals e.g. turning a heater on or turning it off.

A robot arm is an example of a hybrid system with discrete-event input behavior and continuous output behavior. It is controlled through discrete-event signals, controlling e.g. the direction in which to move and it displays continuous behavior, e.g. speed, distance to its destination, or the angle of the arm.

A waferstepper, depicted in Figure 1.1, is an example of a hybrid system with both discrete-event behavior and continuous behavior. A wafer stepper is a machine for manufacturing computer chips. A wafer stepper contains millions of lines of software for controlling e.g. a wafer stage, a vacuum system and a laser. These systems display both discrete-event behavior like communication between software components and continuous behavior like movement of a stage, changing pressure

conditions, and laser light emission.

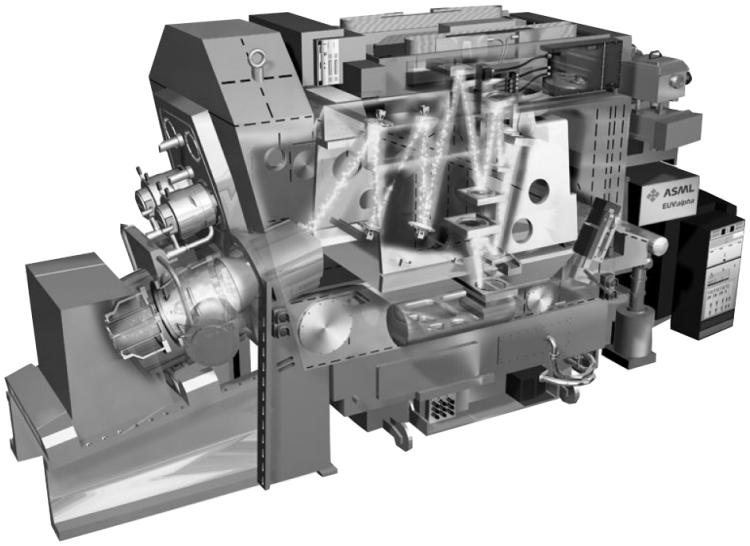

Figure 1.1: A Waferstepper Machine

1.2 Testing

We distinguish three activities in the software and systems engineering process (not taking into account maintenance): designing, implementing and testing. Designing is the activity of making a specification of the system to be implemented. This specification can be very elaborate e.g. made up by user requirements, system requirements, architecture and a detailed design or it can be very concise e.g. made up by some white board drawings and the mental model of the engineers. Implementation is the activity of constructing the system based on the design. Testing is the activity of providing input to a system and observing the output of the system in order to validate whether the behavior of the system conforms to the specification.

A test is a description of which input to apply and when to apply it to the implementation under test, and of which output can be observed from the implementation under test, in order to decide whether the implementation conforms to its design. A test typically consists of the following steps:

- apply an input action and continue the test;
- observe an output action from the implementation and
  - if an output action is observed that was allowed then continue the test,

  - if an output action is observed that was not allowed then terminate the test with verdict fail,
  - if no output action is observed and no output action is supposed to be observed then continue the test, and
  - if no output action is observed but an output action should be observed then terminate with the verdict fail; or

- terminate the test with the verdict pass.

We distinguish three kinds of testing that are used today. The first kind of testing is performing tests without any test description. Examples of this form of testing are ad-hoc testing [1] and exploratory testing [4]. The test engineer determines on the spot which input to provide and, based on the behavior of the system, how to continue. The test engineer thus learns the behavior of the system and based on that information provide new input. E.g. a test engineer might try to apply input he thinks can break the system or tries to apply input that is different from what he already applied before. Based on the mental model of the engineer and the available descriptions of the system it is decided whether the system conforms to the expected behavior.

The second kind of testing is scripted testing. In this case the tests are described by test scripts based on the design of the system. A script can be described in plain text and executed by a test engineer. It can also be described in a scripting language so that it can be executed by a software tool. An example of a scripting language is TTCN-3 [50].

The third kind of testing is model-based testing. In model-based testing a model of the system is used that is based on the design. From this model, test input is automatically generated and executed by a test tool. The output of the system is automatically compared to the output specified by the model of the system. If the system passes all the generated tests, then the system is considered to be correct.

Model-based testing combines the advantages of both other kinds of testing. The model clearly prescribes which input may be applied and which output can be observed It allows exploration of the model in order to generate and execute a larger variety of tests than an engineer can think of in either scripted testing or in unscripted testing. Like using scripts, using models also has the advantage that tests can easily be repeated. If models are already used in other disciplines of model-based engineering like model-based design, then it may be possible to reuse these models for model-based testing and for model-based diagnosis.

Over the years several model-based test tools have been implemented that are able to automatically generate tests for discrete-event systems and timed systems in different ways. Qtronic [14] is a tool that generates TTCN-3 test scrips from UML statecharts [37]. Gotcha-TCBeans [20] is a tool that generates a JavaBean component [27, 36] from a language based on the well known mathematical model of finite state machines. Tests generated from a specification can be executed in the tool kit's runtime testing framework. TorX [55, 6] generates and executes tests at

runtime from e.g. a PROMELA [44] like specification. In TorX, tests are generated according to the formally defined test generation algorithm by Tretmans [48].

1.3 Test Formalization

A formal notion of test makes it possible to design and implement a test algorithm that generates and executes tests according to the definition. This guarantees that the tests are generated in an unambiguous way and that the verdict is interpreted in an unambiguous way.

For a formal test theory as described in this thesis we first need a formal representation for an implementation and a specification. Second, we need to define a conformance relation that tells when a formal implementation conforms to a formal specification. Third, we need to define a notion of test that defines what a test is. Fourth, we need to prove the correctness of the notion of test with respect to the conformance relation by proving that the set of tests which can be generated from a model is sound and exhaustive. The notion of test is sound if only faulty implementations can fail a test. The notion of test is exhaustive if every faulty implementation fails at least one test.

1.4 Input-output Conformance Testing

Input-output conformance testing refers to testing whether a system conforms to a model of the system itself. Input-output conformance testing was first formally defined by Tretmans [48]. In input-output conformance testing, the model is called the specification and the system under test is called the implementation. It is a form of black-box testing. Only the observable input and output behavior of the implementation are considered. The internal (unobservable) behavior of the implementation is not considered.

The conformance relation by Tretmans considers discrete-event behavior. According to Tretmans, an implementation conforms to a specification if after every trace consisting of inputs and outputs, the implementation is only able to produce output that is allowed by the specification.

Example 1.1. *A specification of a coffee machine could describe that after inserting fifty cents into the machine, it has to produce a cup of coffee. After a coffee is produced fifty cents can be inserted again. If an implementation of this machine always produces a cup of coffee after inserting the fifty cents, then it conforms to the specification. If the coffee machine produces an empty cup, the coffee machine does not conform to the specification.*

For the coffee machine as described above, a test could be:

1. *insert fifty cents,*

2. *allow a cup of coffee being produced and conclude fail if only a cup is produced or nothing is produced, and*

3. *in case a cup of coffee is produced, continue the test by inserting another fifty cents.*

Tretmans has defined a formal notion of test and has proved that his tests are sound and exhaustive with respect to his conformance relation [48]. As mentioned in Section 1.2, test generation and execution based on this notion of test are implemented in the TorX test tool [55, 6]. With this tool, tests can be generated from a formal model and executed on an implementation. A number of case studies have been performed with TorX [49].

Based on Tretmans's work, several other input-output conformance theories have been defined. Heerink and Tretmans [22] have defined input-output conformance with refusals and multiple input channels. Refusal in this theory means that, unlike in the original conformance relation by Tretmans, the implementation is allowed to refuse input. An input-output conformance theory for systems with data has been defined by Frantzen, Willemse, and Tretmans [17]. Brandán-Briones and Brinksma [12], Krichen and Tripakis [30], Bohnenkamp [10], and Larsen et al. [33] separately have defined input-output conformance and test generation for timed systems. The theory of Bohnenkamp turned out to be basically the same as that of Brandán-Briones and Brinksma. The difference was that Bohnenkamp decided to take the well known mathematical model of timed automata as formal representation for the implementation and the specification in stead of timed labeled transition systems. The theory defined by Krichen and Tripakis turned out to be the same as theory defined by Larsen et al.

Test generation and execution based on the notion of test by Brandan-Briones and Brinksma, and Bohnenkamp has been implemented in TorX. Test generation and execution based on the notion of test by Krichen and Tripakis has been implemented in the prototype test tool TTG [29]. Krichen and Tripakis performed a case study on a Mars rover controller [9]. Independently the same notion of test has been implemented in the test tool TRON [51]. The TRON tool was developed by Larsen et al. [33] and is based on the Uppaal model checking tool. Larsen et. al. performed an industrial case study on a temperature controller [33].

Krichen and Tripakis [32], and Larsen and Nielson [33] also defined a slightly different conformance relation in which an environment is explicitly used to restrict the input behavior of the specification. This change in the relation is only cosmetic, but it presents a more modular view on the specification.

1.5 Overview of this Thesis

The input-output conformance theories discussed in the previous section do not take the hybrid aspects of an implementation into account. This means that the influence of continuous input behavior on a hybrid system and the correctness of continuous

output behavior of a hybrid system cannot be tested. This thesis presents a theory for input-output conformance testing of hybrid systems and it describes how hybrid model-based testing can be implemented in practice.

1.5.1 Theory Development

In order to define the input-output conformance relation we need a formal representation for the implementation and the specification. Like in the input-output conformance theory by Tretmans and the timed input-output conformance theories, we chose to use hybrid transition systems for this purpose. The concepts of labeled transition systems that are important for this thesis we define in Chapter 2. In this chapter we formally define labeled transition systems, timed labeled transition systems, hybrid transition systems, and related concepts.

We will present a hybrid conformance relation based on the already existing discrete-event and timed input-output conformance relations of Tretmans, Brandán-Briones and Brinksma, and Krichen and Tripakis. In Chapter 3 we define these discrete-event and timed conformance theories.

In chapter 4 we will define our hybrid input-output conformance relations. First we define a conformance for hybrid systems without continuous input behavior. Then, we define a hybrid input-output conformance relation for hybrid systems that require continuous input. This is more complex because continuous input and continuous output take place simultaneously and an input-output conformance relation defines whether the output allowed by the implementation conforms the the output allowed by the specification, with respect to input applied by the specification. Subsequently another hybrid input-output conformance relation is formed in which the specification is separated into a system part and an environment part. The system part of the specification describes the behavior of the implementation under test and the environment describes the behavior of the environment in which the implementation is placed.

It is a matter of personal taste to decide which of the above conformance relation is better. The relation with environment clearly distinguishes the environment of the implementation from the implementation itself. The relation without environment makes explicit that the conformance relation only concerns the continuous output allowed by the implementation with respect to the continuous input applied to the implementation. Furthermore, a test tool implementation in which the input for the implementation is derived from a separate and the output of the implementation is validated with the output allowed by the specification is more complex. The reason is that with a separate environment, input and output need to be synchronized between the specification and the implementation and the environment in stead of a synchronization between the implementation and the specification.

We prove that there is no difference in the two relations considering whether an implementation conforms to a specification. If an implementation does conform to a specification with restricted input behavior, then it does also conform to the specification with a separate environment that specifies the input restriction. And, if an

implementation does not conform to a specification with restricted input behavior, then it also does not conform to the specification with a separate environment that specifies the input restriction.

Also in chapter 4 we define a notion of test for hybrid systems. This notion of test is also based on the notions of test for discrete-event and timed systems. After that, we prove that this notion of test is sound and exhaustive with respect to the hybrid conformance relation.

1.5.2 Tool Development

A prototype version of a hybrid test tool has been implemented [41]. This prototype tool is capable of generating tests from a hybrid χ model [5] and executes tests in real-time. The tool architecture was based on that of TorX. In Chapter 5 we present both the relevant background of the TorX test tool and the hybrid χ language.

In Chapter 6 we describe the design issues and implementation issues involved in developing a hybrid test tool in general, and the implemented prototype tool in particular. In practice however, it is often easier or sufficient, or unavoidable, to apply sampling to the continuous behavior. Therefore, we define a conformance relation and notion of test with sampled continuous behavior.

Furthermore, in Chapter 6 we describe a test generation and execution algorithm for testing hybrid systems. This is algorithm for on-the-fly test generation and execution. While the test is executed it is generated simultaneously. Other issues regarding the test tool implementation described in this chapter are: how to select the test input, when to terminate a test, and how to connect the test tool to an implementation.

1.5.3 Industrial Case Study

This research was conducted in cooperation with the company ASML [3]. ASML is the world leading manufacturer of waferstepper machines (see Figure 1.1). A waferstepper is a machine for the manufacturing of computer chips. It is used to etch an image of the chip layout on a silicon die. This silicon disc is processed into chips later on in the production process. The width of the lines on the image is measured in nanometers and usually multiple (more then 20) lines have to etched on top of each other with nanometer precision.

For the customer losing production time is costly. For the manufacturer selling one machine less per year is costly because one machine costs millions of Euros. This makes it all the more important to have a shorter time to market than the competition but still offer a high quality machine. Therefore a waferstepper needs be tested as thoroughly as possible and as fast as possible.

In Chapter 7 we describe a case study that has been performed with our prototype tool. The case study concerned a vacuum control system which can be used in a waferstepper machine. This is a suitable system to test using hybrid input-output

conformance testing. It observes pressure flow through a sensor and it controls pumps and valves by output actions. The kind of waferstepper we consider has multiple chambers which have to be kept in vacuum by a system of pumps, valves and gauges. This hardware is controlled by a software controller implemented in Labview [28]. A hybrid χ model of a part of the controller has been made. To restrict the input to be applied to the controller, the vacuum hardware (pumps and valves) have been modeled as well. Tests showed faulty behavior in the implementation.

In Chapter 8 we present the conclusions of our work and we present the research areas for future work.

2

Transition Systems and Automata

In order to formalize a test relation, a formal representation of the implementation and the specification are needed. The implementation and specification are viewed as labeled transition systems. This chapter formally defines discrete-event, timed, and hybrid transition systems, which we need further on in this thesis.

For examples it is often easier to use automata. An automaton gives a finite representation of a transition system with infinitely many states and infinitely many transitions. In this chapter also timed automata and hybrid automata are defined. A transition system semantics is given for hybrid automata to show how automata in this thesis actually represent transition systems.

2.1 Transition Systems

A labeled transition system (LTS) is a tuple consisting of a set of states, an initial state, a set of labels, and a transition relation between states. A timed labeled transition system (TLTS) is a labeled transition system where the set of labels contains a set of action labels and a set of time labels. A hybrid transition system (HTS) is a labeled transition system where the set of labels contains a set of action labels and a set of trajectories. In this section these three types of transition systems are defined together with a number of concepts we need later on.

2.1.1 Labeled Transition Systems

In the input-output conformance relation of Tretmans [48] theory both the implementation and the specification are viewed as labeled transition systems.

Definition 2.1. *A* labeled transition system (LTS) *is a tuple* $\mathcal{L} = (S, s_0, L \cup \{\tau\}, \rightarrow)$, *where:*

- S *is a (possibly infinite) set of states;*
- $s_0 \in S$ *is the initial state;*
- L *is a (possibly infinite) set of labels; and*
- $\rightarrow \subseteq S \times L \cup \{\tau\} \times S$ *is the transition relation.*

A transition with a special label τ represents an internal event. Instead of $(s, \mu, s') \in \rightarrow$ we usually write $s \xrightarrow{\mu} s'$.

The transition relation defines a set of transitions from one state to another state. These transitions describe the behavior performed by a system. We say a state allows a certain transition if there is a transition from that state to another state.

The partial execution of a transition system is described as allowing a sequence of transitions. We say a transition system can perform transition t after a sequence $\alpha \in L \cup \{\tau\}^*$ if, after the execution of α, it allows t.

An input-output conformance relation is a relation on the input-output behavior of the specification and the implementation. This behavior is also called the *observable* behavior of the system. The internal behavior is not taken into account. This is restricted by means of a generalized transition relation.

Definition 2.2. *Let $\mathcal{L} = (S, s_0, L \cup \{\tau\}, \rightarrow)$ be an LTS and let $s, s', s'' \in S$ be states. Let ϵ denote the empty sequence. The* generalized transition relation *is the least relation $\Rightarrow \subseteq S \times L^* \times S$ such that:*

- *$s \overset{\epsilon}{\Rightarrow} s$;*
- *if $s \xrightarrow{\tau} s'$, then $s \overset{\epsilon}{\Rightarrow} s'$;*
- *for all $l \in L$, if $s \xrightarrow{l} s'$, then $s \overset{l}{\Rightarrow} s'$; and*
- *for all $\alpha, \beta \in L^*$, if $s \overset{\alpha}{\Rightarrow} s''$ and $s'' \overset{\beta}{\Rightarrow} s'$ then $s \overset{\alpha\beta}{\Rightarrow} s'$.*

From now on, $s \xrightarrow{l}$ denotes that there is a state s' such that $s \xrightarrow{l} s'$, and $s \overset{\alpha}{\Rightarrow}$ denotes that there is a state s' such that $s \overset{\alpha}{\Rightarrow} s'$. Furthermore, $s \overset{l}{\not\rightarrow}$ denotes that there is no state s' such that $s \xrightarrow{l} s'$, and $s \overset{\alpha}{\not\Rightarrow}$ denotes that there is no state $s' \in S$ such that $s \overset{\alpha}{\Rightarrow} s'$.

An LTS is *deterministic* if from a state it allows more than one transition with the same label leading to different states.

Definition 2.3. *An LTS $\mathcal{L}$ is* deterministic *if, for all $s, s', s'' \in S$ and $l \in L$, if $s \overset{l}{\Rightarrow} s'$ and $s \overset{l}{\Rightarrow} s''$, then $s' = s''$.*

In this thesis we only consider finite sequences of labels. A sequence of labels formed by performing a sequence of transitions from the initial state, with the internal actions removed, is called a trace.

Definition 2.4. *A* trace *is a finite sequence of labels $\alpha \in L^*$ such that $s_0 \overset{\alpha}{\Rightarrow}$. We denote by $traces(\mathcal{L})$ the set of all traces of $\mathcal{L}$, i.e. $traces(\mathcal{L}) = \{\alpha | s_0 \overset{\alpha}{\Rightarrow}\}$.*

For the purpose of defining an input-output conformance relation we partition the set of labels L into a set of input actions A_I, a set of output actions A_O and the internal action τ. The set of labels consisting of the set of actions in A_I together with the set of actions in A_O is denoted by the set of actions A, i.e. $A = A_I \uplus A_O$.

A_τ denotes the set $A \cup \{\tau\}$. Note that $\uplus$ denotes the union of two disjunct sets, e.i. $A_I \uplus A_O$ denotes the union of the disjunct sets A_I and A_O.

In input-output conformance testing by Tretmans [48] an implementation is assumed to be input enabled. An LTS is *input enabled* if it can accept any input at any moment in time. In the LTSs considered so far the notion of time is not made explicit. Therefore, an LTS is *input enabled* if in every state, every input action is allowed.

Definition 2.5. *An LTS* $= (S, s_0, L \cup \{\tau\}, \rightarrow)$ *with* $L = A_I \uplus A_O$ *is* input enabled *if every input action is possible in every state; that is: for all* $s \in S$ *and* $i \in A_I$*:* $s \xrightarrow{i}$.

It will be convenient to use process algebraic notation for LTSs. This notation will be useful when we formally define the notion of test. If $\mathcal{L}$ is an LTS and l is a label, then by $l;\mathcal{L}$ we denote the LTS that from the initial state allows the transition with label l only, and after that behaves as $\mathcal{L}$. If $\mathcal{LS}$ is a set of LTSs, then by $\sum \mathcal{LS}$ we denote the LTS that in its initial state allows the choice to perform one of the LTSs in $\mathcal{LS}$. If $\mathcal{L}_1$ and $\mathcal{L}_2$ are two LTSs then by $\mathcal{L}_1 \parallel \mathcal{L}_2$ we denote the LTS that can perform an observable action if it can be performed by both LTSs and that can perform an internal action if it can be performed by either LTS.

Definition 2.6. *Let* $\mathcal{L} = (S, s_0, L \cup \{\tau\}, \rightarrow)$ *be an LTS. Let* $s \notin S$ *be a fresh state. The result of* prefixing $\mathcal{L}$ *with a transition with label* l*, denoted by* $l;\mathcal{L}$*, is defined by*

$$l;\mathcal{L} = (S \cup \{s\}, s, L \cup \{\tau\} \cup \{l\}, \rightarrow \cup\ \{(s, l, s_0)\}).$$

Definition 2.7. *Let* $\mathcal{LS} = \{\mathcal{L}_i | i \in I\}$ *be a (possibly infinite) set of LTSs of the form* $\mathcal{L}_i = (S_i, s_{0i}, L_i \cup \{\tau\}, \rightarrow_i)$*. Let* $s \notin \bigcup_{i \in I} S_i$ *be a fresh state.*

- *The result of the* alternative composition *of the set of LTSs* $\mathcal{LS}$*, denoted by* $\sum \mathcal{LS}$*, is defined by*

  $$\sum \mathcal{LS} = (\bigcup_{i \in I} S_i \cup \{s\}, s, \bigcup_{i \in I} L_i \cup \{\tau\}, \bigcup_{i \in I} (\rightarrow_i \cup \{(s, l, s') | s_{0i} \xrightarrow{l}_i s'\})).$$

  Furthermore, $\mathcal{L}_0 + \mathcal{L}_1$ *denotes* $\sum\{\mathcal{L}_0, \mathcal{L}_1\}$.

- *The* synchronous composition *of* $\mathcal{L}_0$ *and* $\mathcal{L}_1$*, denoted by* $\mathcal{L}_0 \parallel \mathcal{L}_1$ *is defined by*

  $$\mathcal{L}_0 \parallel \mathcal{L}_1 = (S_1 \times S_2, (s_{00}, s_{01}), L_0 \cap L_1, \rightarrow)$$

  with $\rightarrow =$

  $$\{((s_0, s_1), l, (s_0', s_1')) | s_0 \xrightarrow{l}_0 s_0' \wedge s_1 \xrightarrow{l}_1 s_1' \wedge l \in (L_0 \cap L_1) \setminus \{\tau\}\} \cup$$
  $$\{((s_0, s_1), \tau, (s_0, s_1')) | s_1 \xrightarrow{\tau}_1 s_1'\}\ \cup\ \{((s_0, s_1), \tau, (s_0', s_1)) | s_0 \xrightarrow{\tau}_0 s_0'\}.$$

Note that the definition for alternative composition also works LTSs that are *root cyclic*. An LTS is *root cyclic* if there is a nontrivial path from the initial state to itself.

2.1.2 Timed Labeled Transition Systems

A timed labeled transition system is a labeled transition system with action labels and time labels. A time label is a nonnegative real number. It denotes an amount of time that can pass. We also call these labels durations. Time does not pass by performing actions.

Definition 2.8. *A* timed labeled transition system *(TLTS)* $\mathcal{T} = (S, s_0, L \cup \{\tau\}, \rightarrow)$ *is an LTS with the set of labels partitioned into a set of action labels and a set of durations in* $\mathbb{R}^{\geq 0}$*:* $L = A_\tau \uplus \mathbb{R}^{\geq 0}$*. We write* $s \xrightarrow{a(t)} s''$ if there is an $s' \in S$ such that $s \xrightarrow{t} s'$ and $s' \xrightarrow{a} s''$.

Three constraints are generally imposed on TLTSs:

- **C1 (null delay):** for all $s, s' \in S$, $s \xrightarrow{0} s'$ if and only if $s = s'$.
- **C2 (time additivity):** for all $t_1, t_2 > 0$ and $s, s'' \in S$, $s \xrightarrow{t_1+t_2} s''$ if and only if there exists an $s' \in S$ such that: $s \xrightarrow{t_1} s'$ and $s' \xrightarrow{t_2} s''$.
- **C3 (time determinism):** for all $s, s', s'' \in S$ and $t_1 \in \mathbb{R}^{\geq 0}$, if $s \xrightarrow{t_1} s'$ and $s \xrightarrow{t_1} s''$ then $s' = s''$.

Constraint **C1** states that the state does not change by 0 delay. Constraint **C2** states that waiting some t_1 time units and after that waiting t_2 time units more is the same as waiting $t_1 + t_2$ time units and waiting time $t_1 + t_2$ time units is the same as waiting first t_1 time units and then waiting t_2 time units. Constraint **C3** states that letting time pass does not make a choice on the state that is reached after that time has passed.

A TLTS is input enabled if at every moment in time every input action is allowed. Brandán-Briones and Brinksma [12] use a definition in which every input action is eventually allowed after some internal activity.

Definition 2.9. *A TLTS* $\mathcal{T}$ *is* input enabled *if, for all* $s \in S$ *and* $i \in A_I$*,* $s \stackrel{i}{\Rightarrow}$.

Again it will be convenient to introduce process algebraic notation for TLTSs. The definitions of synchronous composition and prefixing an TLTS with an action are the same as for LTSs. When defining the notion of prefixing an LTS with a duration and the alternative composition of a set of LTSs we need to take care to maintain the constraints **C1** to **C3**. If a TLTS $\mathcal{T}$ is prefixed by a duration t, then the resulting TLTS, denoted by $t;\mathcal{T}$, allows all time transitions up to the duration t and after that behaves like $\mathcal{T}$. If a set of TLTSs $\mathcal{TS}$ is alternatively composed, then the resulting TLTS, denoted by $\sum \mathcal{TS}$, models the choice to perform one of the TLTSs in TS. The resulting TLTS may be non-deterministic in the actions but should be time deterministic. This means that the final choice of which TLTS is executed is only made at the moment that the first action is performed.

Definition 2.10. *Let* $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ *be a TLTS and let* $t \in \mathbb{R}^{\geq 0}$ *be a duration, then the result of prefixing* $\mathcal{T}$ *with a time transition with duration* t*,*

denoted by $t;\mathcal{T}$ *is defined as:*

$$t;\mathcal{T} = (S \uplus S', s_t, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow \cup \rightarrow'),$$

where

$$S' = \{s_{t'} | 0 < t' \leq t\},$$

and

$$\begin{aligned} \rightarrow' = \{&(s_{t'}, t' - t'', s_{t''}) | 0 \leq t'' < t' \leq t\} \cup \\ &\{(s_{t'}, t' + t'', s') | 0 < t' \leq t \wedge s_0 \xrightarrow{t''} s'\}. \end{aligned}$$

Definition 2.11. *Let* $\mathcal{TS} = \{\mathcal{T}_i | i \in I\}$ *be a (possibly infinite) set of TLTSs of the form* $\mathcal{T}_i = (S_i, s_{0i}, A_i \cup \mathbb{R}^{\geq 0} \cup \{\tau\}, \rightarrow_i)$*. Let* $s \notin \bigcup_{i \in I} S_i$ *be a fresh state. The* alternative composition *of* $\mathcal{TS}$*, described by* $\sum \mathcal{TS}$*, is defined by*

$$\sum \mathcal{TS} = (\bigcup_{i \in I} S_i \cup S' \cup \{s\}, s, \bigcup_{i \in I} A_i \cup \mathbb{R}^{\geq 0} \cup \{\tau\}, \bigcup_{i \in I} (\rightarrow_i \cup \rightarrow')).$$

where

$$S' = \{s_t | t \text{ is a duration such that } s_{0i} \xrightarrow{t} \text{ for some } i \in I\},$$

and

$$\begin{aligned} \rightarrow' = \{&(s, a, s') | \exists_{i \in I} : s_{0i} \xrightarrow{a}_i s'\} \cup \\ &\{(s_t, a, s') | \exists_{i \in I} : s_{0i} \xrightarrow{a(t)}_i s'\} \cup \\ &\{(s, t, s_t) | s_t \in S'\} \cup \\ &\{(s_t, t'', s_{t'}) | s_t, s_{t'} \in S' \wedge t + t'' = t'\}. \end{aligned}$$

It is assumed in the definitions above that $(\bigcup_{i \in I} S_i) \cap (S' \cup \{s\}) = \emptyset$ *Furthermore,* $\mathcal{T}_0 + \mathcal{T}_1$ *denotes* $\sum\{\mathcal{T}_0, \mathcal{T}_1\}$.

2.1.3 Hybrid Transition Systems

The definition of hybrid transition systems presented in this section is based on the definition given by Cuijpers, Reniers, and Heemels [15] together with the definition of trajectories given by Lynch, Segala and Vaandrager [34]. A hybrid transition system is able to describe both the discrete-event behavior and the continuous behavior of a system. For instance, it allows to describe a heater that, if it is turned on, heats a room with 0.1 ^{o}C/min. In a hybrid transition system, the continuous behavior is described by trajectories. A *trajectory* describes the evolution of a set of continuous variables over an interval of time. Time does not pass by performing actions. The evolution of continuous variables is also called *flow*. Trajectories in our theory are continuous in the sense that for every element in the interval of a trajectory, a valuation of variables is defined. However they are allowed to have discontinuities in the sense that jumps in the valuations are allowed, e.i a trajectory is not necessarily differentiable.

Definition 2.12. *An interval over* $\mathbb{R}$ *is called* left-closed *if it has a minimum and it is called* right-closed *if it has a maximum. It is called* closed *if it has both a minimum and a maximum. An interval is called* left-open *if it has no minimum and it is called* right-open *if it has no maximum. It is called* open *if it has no minimum and no maximum.*

According to Lynch, Segala, and Vaandrager [34], a *valuation* on a set of variables V is a function that associates with each variable $v \in V$ a value of the type of v. In this thesis we consider only variables of type real. We write $Val(V)$ for the set of all valuations on V. A trajectory is a function that associates a valuation of variables with each element in a time interval.

Definition 2.13. *Let* V *be a set of variables. A trajectory* σ *is a function* $\sigma : D \to Val(V)$ *that associates with each element in the interval* D *a valuation. We write trajs(V) for the set of all trajectories with respect to* V*.*

Unless otherwise stated, the domain D of a trajectory is a left-open right-closed interval starting at (but not including) 0. A one side open time interval ensures that subsequent trajectories do not overlap each other. Contrary to Cuijpers, Reniers, and Heemels [15] we choose right-closed time intervals in order to reason about the value of variables after a trajectory.

We need a number of operations and shorthand notations on trajectories.

Definition 2.14.

1. *Let* $f : A \to B$ *be a function; for* $A' \subseteq A$ *we define* f *restricted to* A'*, denoted by* $f\lceil A'$*, as the function* $f\lceil A' : A' \to B$ *defined by* $f\lceil A'(a) = f(a)$ *for* $a \in A'$*.*

2. *Let* $f : A \to B$ *with* $A \subseteq \mathbb{R}$ *be a function and let* $A + t = \{a + t | a \in A\}$*; for* $t \in \mathbb{R}$ *we define* $f + t : A + t \to B$ *by* $(f + t)(t') = f(t' - t)$*, with* $t' \in A + t$*.*

3. *Let* σ *be a trajectory. We write* $dom(\sigma)$ *for the domain of* σ*,* $\sigma.lval$ *for the last valuation of* σ*, and* $\sigma.ltime$ *for the maximum of the domain of* σ*. Note that* $\sigma.lval = \sigma(\sigma.ltime)$*.*

4. *If* f *is a function on a closed interval, then We write* $f.fval$ *for the first valuation of* f *and we write* $f.ftime$ *for the minimum of the domain of* f*.*

5. *Let* σ *be a trajectory on variables* V*; for* $V' \subseteq V$ *we define* σ *restricted to* V'*, denoted by* $\sigma \downarrow V'$*, as the function* $\sigma \downarrow V' : D \to Val(V')$ *defined by* $\sigma \downarrow V'(t) = \sigma(t)\lceil V'$ *for* $t \in D$*.*

6. *Let* σ *and* σ' *be trajectories. The concatenation of* σ *and* σ' *(denoted by* $\sigma \frown \sigma'$*) is defined as:*
$$\sigma \frown \sigma' \;=\; \sigma \cup (\sigma' + \sigma.ltime).$$

7. *Let* σ *be a trajectory defined over an interval* $(0, t]$*, with* $t > 0$ *and let* $t' \in \mathbb{R}^{>0}$ *with* $t' \leq t$*; then:*
$$\begin{aligned} \sigma \trianglelefteq t' &= \sigma\lceil(0, t']; \\ \sigma \trianglerighteq t' &= (\sigma\lceil(t', t]) - t'. \end{aligned}$$

8. *Let σ and σ' be two trajectories; then σ is a prefix of σ', denoted by $\sigma \leq \sigma'$, if there exists a $t \in \mathbb{R}^{>0}$ such that $\sigma = \sigma' \trianglelefteq t$. We write $\sigma < \sigma'$ if $\sigma \leq \sigma'$ and $\sigma \neq \sigma'$.*

A hybrid transition system is a labeled transition system with action labels and trajectory labels.

Definition 2.15. *A hybrid transition system (HTS) $\mathcal{H} = (S, s_0, L, \rightarrow)$ is an LTS with the set of labels L partitioned into a set of action labels A and a set of trajectories Σ: $L = A_\tau \uplus \Sigma$. For the set of trajectories Σ we impose that $\sigma_1 \frown \sigma_2 \in \Sigma$ if and only if $\sigma_1 \in \Sigma$ and $\sigma_2 \in \Sigma$.*

To make the distinction between discrete-event transitions and continuous transitions clearer, we write $s \overset{\sigma}{\rightsquigarrow} s'$ for a transition with a trajectory label instead of $s \xrightarrow{\sigma} s'$. In case a label l does not fit syntactically above a continuous transition, we write $s \overset{l}{\rightsquigarrow} s'$ instead of $s \overset{l}{\rightsquigarrow} s'$.

Like for TLTSs, an additivity condition and a determinism condition are imposed on HTSs. A null delay condition is not needed because trajectories always have a left-open domain.

- **A1 (trajectory additivity):** for all $\sigma, \sigma' \in \Sigma$ and $s, s'' \in S$, $s \overset{\sigma' \frown \sigma''}{\rightsquigarrow} s''$ if and only if there exists an $s' \in S$ such that: $s \overset{\sigma'}{\rightsquigarrow} s'$ and $s' \overset{\sigma''}{\rightsquigarrow} s''$.
- **A2 (trajectory determinism):** for all $\sigma \in \Sigma$ and $s, s', s'' \in S$, if $s \overset{\sigma}{\rightsquigarrow} s'$ and $s \overset{\sigma}{\rightsquigarrow} s''$ then $s' = s''$.

Constraint **A1** is necessary for the conformance theory. Otherwise, an implementation that first performs a trajectory for 2 seconds followed by performing a trajectory for 3 seconds is not conform a specification that performs the same flow of continuous variables for 5 seconds.

For a HTS $\mathcal{H}$ with the set of trajectories Σ defined on a set of variables V, V is always partitioned into a set of input variables V_I, and a set of output variables V_O. A HTS (with $L = A_\tau \uplus \Sigma$) is *input enabled* if it allows every input action at every moment in time and every flow of input variables at every moment in time.

Definition 2.16. *A HTS $\mathcal{H}$ is* input enabled *if:*

- *for every $s \in S$ and $i \in A_I$, $s \overset{i}{\Rightarrow}$; and*
- *for every $s \in S$:*
  1. *there exists an action $a \in A_O \cup \{\tau\}$ such that $s \xrightarrow{a}$ and there does not exist a $\sigma \in \Sigma$ such that $s \overset{\sigma}{\rightsquigarrow}$; or*
  2. *for every $u \in trajs(V_I)$ there exists a $\sigma \in \Sigma$ with $\sigma \downarrow V_I \leq u$ and:*
     (a) $\sigma \downarrow V_I = u$ and there exists an $s' \in S$: $s \overset{\sigma}{\rightsquigarrow} s'$; or

(b) $\sigma \downarrow V_I < u$ *and there exists an action* $a \in A_O \cup \{\tau\}$ *and* $s' \in S$ *such that:* $s \overset{\sigma}{\rightsquigarrow} s' \overset{a}{\rightarrow}$.

This definition is adapted from the definition of input enabledness as it was defined by Lynch, Segala and Vaandrager [34] for hybrid input-output automata. It states that in every state all input actions have to be allowed, possibly preceded by internal actions. In addition, in every state either no trajectories are allowed or it allows every continuous behavior for the input variables, possibly interrupted by an output action or an internal action. The interruption does not disturb the input enabledness because after a sequence of interrupts also every continuous input is accepted, including the suffix of the interrupted continuous input. An interrupt by output action or internal action does not disturb the input enabledness, since these actions do not take time and after the interrupt, again all continuous input is allowed, including the remainder of the interrupted continuous input.

Like for LTSs and TLTSs it will be convenient to introduce process algebraic notation for HTSs. The definitions of synchronous composition and prefixing an HTS with an action are the same as for LTSs. The definitions for prefixing a HTS with a trajectory and the alternative composition of a set of HTSs need to be reconsidered for HTSs because the constraints **A1** and **A2** need to be maintained. If a HTS $\mathcal{H}$ is prefixed by a trajectory σ, then the resulting HTS, denoted by $\sigma;\mathcal{H}$, first behaves according to the trajectory σ and after that behaves like $\mathcal{H}$. If a set of HTSs $\mathcal{HS}$ is alternatively composed, then the resulting HTS, denoted by $\sum \mathcal{HS}$, models the choice to perform one of the HTSs in $\mathcal{HS}$. The resulting HTS may be non-deterministic in the actions but is trajectory deterministic. This means that the final choice is only made at the moment that the first action is performed.

Definition 2.17. *Let* σ *be a trajectory, let* $r = \sigma.ltime$, *and let* $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ *be a HTS. Then* $\sigma;\mathcal{H}$, *the HTS that results from prefixing* $\mathcal{H}$ *with the trajectory* σ, *is defined by*

$$\sigma;\mathcal{H} = (S \uplus S', s_r, L', \rightarrow \cup \rightsquigarrow \cup \rightsquigarrow') \ ,$$

where

$$S' = \{s_{r'} \mid r \geq r' > 0\},$$

and

$$L' = A_\tau \cup \Sigma \ \cup \{\sigma | (\sigma \trianglerighteq (r - r')) \trianglelefteq (r' - r'') \wedge r \geq r' > r'' \geq 0\} \cup \{\sigma | (\sigma \trianglerighteq (r - r')) \frown \sigma' \wedge r \geq r' > 0\}$$

and

$$\rightsquigarrow' = \{(s_{r'}, (\sigma \trianglerighteq (r - r')) \trianglelefteq (r' - r''), s_{r''}) \mid r \geq r' > r'' \geq 0 \wedge s_{r'}, s_{r''} \in S'\} \cup \{(s_{r'}, (\sigma \trianglerighteq (r - r')) \frown \sigma', s') \mid r \geq r' > 0 \ \wedge s_0 \overset{\sigma'}{\rightsquigarrow} s' \wedge s_{r'} \in S'\}.$$

(It is assumed in the definition that $s_{r'} = s_{r''}$ *implies* $r' = r''$ *for all* $r \geq r', r'' > 0$.*)*

Definition 2.18. *Let* $\mathcal{HS} = \{\mathcal{H}_i \mid i \in I\}$ *with* $\mathcal{H}_i = (S_i, s_{0i}, L_i \cup \{\tau\}, \rightarrow_i \cup \rightsquigarrow_i)$ *be a set of hybrid transition systems. We define the summation* $\sum \mathcal{HS}$ *by*

$$\sum \mathcal{HS} = (\bigcup_{i \in I} S_i \uplus S' \uplus \{s_0\}, s_0, \bigcup_{i \in I} L_i \cup \{\tau\}, \bigcup_{i \in I} \rightarrow_i \cup \rightarrow' \cup \bigcup_{i \in I} \rightsquigarrow_i \cup \rightsquigarrow'),$$

where

$$S' = \{s_\sigma \mid \sigma \text{ a trajectory s.t. } s_{0i} \overset{\sigma}{\rightsquigarrow} \text{ for some } i \in I\},$$

$$\rightarrow' = \{(s_0, a, s') \mid \exists i \in I.\ s_{0i} \overset{a}{\rightarrow}_i s'\} \cup \{(s_\sigma, a, s') \mid \exists i \in I, s''.\ s_{0i} \overset{\sigma}{\rightsquigarrow}_i s'' \overset{a}{\rightarrow}_i s'\},$$

and

$$\rightsquigarrow' = \{(s_0, \sigma, s_\sigma) \mid s_\sigma \in S'\} \cup \{(s_\sigma, \sigma'', s_{\sigma'}) \mid s_\sigma, s_{\sigma'} \in S' \wedge \sigma \frown \sigma'' = \sigma'\}.$$

Furthermore, we write $\mathcal{H}_0 + \mathcal{H}_1$ *instead of* $\sum\{\mathcal{H}_0, \mathcal{H}_1\}$.

2.2 Automata

To illustrate the principles of input-output conformance testing in the case of timed systems and hybrid systems it is often convenient to use automata instead of transition systems. The reason is that automata are easier to represent graphically. In this section we define the syntax and hybrid transition system semantics of hybrid automata [23]. For us, timed automata [8] are hybrid automata in which all the variables have constant flow 1.

Definition 2.19. *A hybrid automaton* $\mathcal{HA}$ *is a tuple* $(Loc, (l_0, v_0), V, A_\tau, \rightarrow, I, F)$ *, where:*

- *Loc is a finite set of locations;*
- l_0 *is the initial location and* v_0 *is the initial valuation of variables, together they form the initial state;*
- V *is the set of variables; by* $\dot{V}$ *we denote the set of dotted variables of* V*, which represent the first derivatives during continuous change of the variables in* V*;*
- A_τ *is a set of action labels, including the internal action* τ*;*
- $\rightarrow \subseteq Loc \times \mathcal{B}(V \cup \dot{V}) \times A \times PVal(V) \times Loc$ *is the set of switches where:*
  - $\mathcal{B}(V)$ *is a set of boolean constraints on variables and dotted variables that serve as guards to the switches; and*
  - $PVal(V) = \bigcup_{V' \subseteq V} Val(V')$ *is the set of variable assignments, also called resets, that assign a value to (a subset of) the variables* V*;*
- $I : Loc \rightarrow \mathcal{B}(V)$ *is a function that assigns constraints on variables to locations that serve as invariants for the locations.*
- $F : Loc \rightarrow \mathcal{B}(V \cup \dot{V})$ *is a function that assigns constraints to locations that describe the continuous behavior in the location.*

We write $l \xrightarrow{g,a,r} l'$ for $(l, g, a, r, l') \in \rightarrow$.

We assume that F does not allow continuous behavior that does not satisfy axioms **A1** and **A2** on HTSs. If, for all $l \in Loc$, $F(l)$ is a set of ordinary differential equations and piecewise differential equations, like in the examples in this thesis, this is the case.

Before we can define the HTS associated with a HA we need to define two operators. First, we need to denote that a valuation satisfies an invariant in a location or a guard of a switch. Second, we need to define setting a subset of variables of a valuation with the variable assignment on a switch.

Definition 2.20. *Let $\mathcal{HA} = (Loc, (l_0, v_0), V, A_\tau, \rightarrow, I, F)$ be a hybrid automaton.*

- *Let $u \in Val(V)$ be a valuation on variables V and let $l \in Loc$ be a location, then $u \in I(l)$ denotes that the invariant $I(l)$ holds for u and $u \in g$ denotes that the guard g holds for u.*
- *Let $u \in Val(V)$ be a valuation on variables V and let $r \in PVal(V)$ be a reset on a set of variables $V' \subseteq V$. Then by u/r we denote the valuation that behaves as r on V' and as u outside V', i.e.*

$$u'(v) = \begin{cases} r(v) \text{ , if } v \in V'; \text{ and} \\ u(v) \text{ , otherwise.} \end{cases}$$

Henzinger [23] defined a timed transition system semantics for hybrid automata. We define a hybrid transition system semantics for hybrid automata. The difference with transition system semantics for hybrid automata as defined by Henzinger is that in his definition, the (continuous) transitions are labeled with the durations of trajectories while we use trajectories as labels.

A hybrid automaton is defined as a hybrid transition system in which:

- the set of states consists of pairs of locations of the automaton and valuations of variables;
- the initial state is a pair that consists of the initial location of the automaton and the initial valuation of the variables;
- the set of trajectories consists of all flows that, for each location of the automaton, behave according to the flow conditions and the invariants;
- the set of transitions on actions consists of all transitions between states with a different location, for which the set of valuations of start state satisfy the guards of a switch in the automaton, and the set of valuations have a new value in accordance with the reset defined for the switch; and
- the set of transitions on trajectories consists of all flows of each location, from (but not including) the start valuation to the end valuation.

Definition 2.21. *A hybrid automation* $\mathcal{HA} = (Loc, (l_0, v_0), V, A_\tau, \rightarrow, I, F)$ *defines the hybrid transition system* $\mathcal{H} = (Loc \times Val(V), (l_0, u_0), A_\tau \cup \Sigma_F, \rightarrow \cup \rightsquigarrow)$ *where:*

- Σ *is the set of all left-open right-closed trajectories for which there exists an* $l \in L$ *such that for all* $\sigma \in \Sigma$, σ *satisfies* $F(l)$ *and, for all* $t \in dom(\sigma)$, $\sigma(t) \in I(l)$.
- $\rightarrow =$

$$\{(l,u) \xrightarrow{a} (l',u') | l \xrightarrow{g,a,r} l' \wedge u \in g \wedge u \in I(l) \wedge u' = u/r \wedge u' \in I(l')\};$$

and

- $\rightsquigarrow =$

$$\{(l,u) \overset{\sigma}{\rightsquigarrow} (l,u') | l \in Loc \wedge \exists_{f \in \Sigma_F} : u = f.fval \wedge u' = f.lval \wedge dom(\sigma) = (0, f.ltime] \wedge \forall_{t \in dom(\sigma)} : \sigma(t) = f(f.ftime + t)\}$$

where, Σ_F *is the set of all trajectories on a closed interval such that, for all* $f \in \Sigma_F$, f *satisfies* $F(l)$ *and, for all* $t \in dom(f)$, $f(t) \in I(l)$.

2.3 Example Models

The input-output conformance theories presented in this thesis are illustrated by labeled transition systems, fragments of hybrid transition systems, timed automata and hybrid automata. For all of these models we use a graphical notation. In this section we informally describe how to interpret these examples.

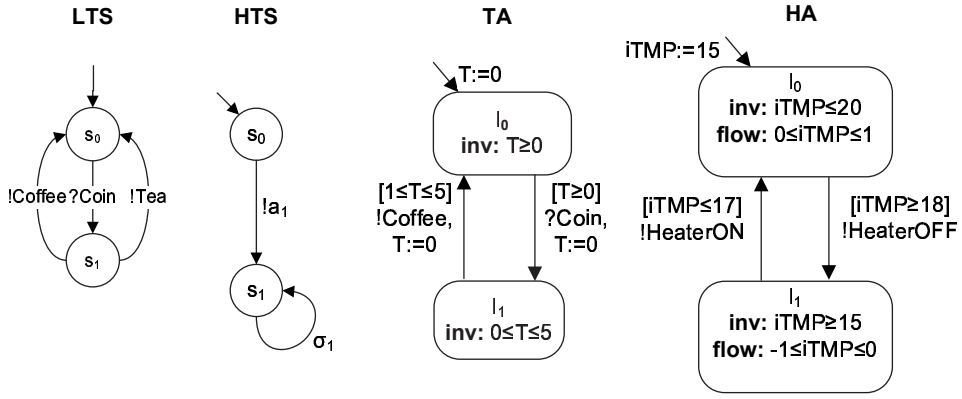

Figure 2.1: An example LTS, HTS, TA, and HA

The LTS in Figure 2.1 models a coffee machine that can receive coins and returns coffee or tea. After a coffee or tea is returned a new coin can be inserted. It has two states s_0 and s_1. The initial state is s_0, which is indicated by the arrow on top.

The action ?Coin is considered an input action; this is indicated by the question mark. The action !Coffee and the action !Tea are output actions; this is indicated by the exclamation mark. The transitions are $(s_0, ?\text{Coin}, s_1)$, $(s_1, !\text{Coffee}, s_0)$, and $(s_1, !\text{Tea}, s_0)$.

The HTS fragment in Figure 2.1 models two states s_0 and s_1. This is only a fragment of an HTS because it does not satisfy the constraints **A1** and **A2** which we posed on HTSs. It has one output action and the discrete-event transition $(s_0, !a_1, s_1)$. It has one trajectory σ_1 and the continuous transition (s_1, σ_1, s_1).

The TA in Figure 2.1 also models a coffee machine. It can receive a coin at any time after which it returns a coffee after a minimum waiting time of 1 second but within 5 seconds. After that a new coin can be inserted. It has two locations l_0 and l_1. The location l_0 is the initial location, which is indicated by the arrow on top. The variable T is called a clock variable. Initially, T is set to 0. In a timed automaton the flow rate of the clock variables are not depicted (because they are always 1). The location invariant of l_0 is $T \geq 0$. This means the automaton can stay forever in this location. The guards of the discrete-event transitions are depicted within square brackets. If the guard is omitted, then the guard is assumed to be $[true]$. The transition $(l_0, T \geq 0, ?\text{Coin}, T := 0, l')$ can be taken at any time and when this transition is taken the clock T is reset to 0. If the reset is omitted, then all variables have the same valuation after the transition as before the transition. The invariant of l_1 is $0 \leq T \leq 5$ which means that the automaton in this example can stay in this location for a maximum of 5 time units (since the clock variable T was reset to value 0. The guard on the transition $(l_0, 1 \leq T \leq 5, !\text{Coffee}, T := 0, l')$ indicates that this transition has to take place within 5 time units as well. Note that if both the invariant and the guard of the outgoing transition of a TA are false, or if the invariant of the location after the transition is false, then time stops and the automaton deadlocks.

The HA in Figure 2.1 models a thermostat. The variable $iTMP$ models the temperature input of the thermostat. We indicate that a variables models continuous input by prefixing the the name of the variable by an 'i' and we indicate that a variable models continuous output by prefixing the the name of the variable by an 'o' respectively. This thermostat keeps the temperature of a room between 15 ^oC and 20 ^oC. The initial location is l_0, which is indicated by the arrow on top of the location. Initially the room is assumed to have a temperature of 15 ^oC. In location l_0 the heater is on and the thermostat accepts temperature increase between 0 ^oC and 1 ^oC per unit of time. Note that therefore this automaton is not input enabled. With a different temperature change it deadlocks. The location invariant is $iTMP \leq 20$, which means that the heater needs to be turned off by the time that the temperature reaches 20 ^oC. The heater can be turned off by the transition to l_1 which is allowed by the guard if the temperature exceeds 18 ^oC. If the heater is off, then the thermostat accepts temperature decrease between -1 ^oC and 0 ^oC per unit of time. If the decreasing temperature goes below 17 ^oC the heater may be turned on again by performing the transition with the label !HeaterON.

3

Discrete-event and Timed Input-output Conformance and Tests

The hybrid input-output conformance theory described in this thesis is based on existing conformance theories. It is based on the discrete-event input-output conformance theory by Tretmans [48], the timed input-output conformance theory by Brandán-Briones and Brinksma [12], and the timed input-output conformance theory by Krichen and Tripakis [30]. This chapter explains those theories in order to give insight in the issues involved in forming an input-output conformance relation and a notion of tests, and to relate hybrid input-output conformance and tests to these theories later on.

3.1 Discrete-event Input-output Conformance and Tests

In conformance testing, tests are derived from a specification and if all tests lead to the verdict pass, then the implementation conforms to the specification. The implementation is assumed to be input enabled. This is a reasonable assumption because it should be possible to test the reaction of the implementation on any input at any moment in time. The specification does not need to be input enabled. This allows to test the reaction of the implementation for only part of its behavior.

Tretmans [48] has given a formalization of what conformance between a specification and an implementation means and he has given a formalization of tests. In his theory, both the implementation and the specification are viewed as LTSs. Tretmans defines that an input enabled implementation conforms to a specification if after every trace of input actions and output actions of the specification, the output actions allowed by the implementation form a subset of the output actions allowed by the specification. This exactly means that if the implementation can perform an output action that is not specified, then the implementation does not conform to the specification.

Example 3.1. *Figure 3.1 depicts a specification and two implementations of a coffee machine in the form of labeled transition systems. These implementations are input*

enabled with respect to inserting a coin, since in every state they accept the coin.

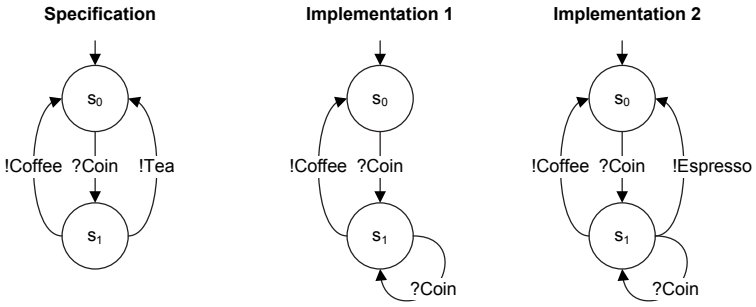

Figure 3.1: An example coffee machine

The specification of the coffee machine repetitively accepts a coin and produces a coffee or a tea. Implementation 1 repetitively accepts one or more coins and produces a coffee. Implementation 2 repetitively accepts one or more coins and produces a coffee or an espresso.

Implementation 1 conforms to the specification. According to the specification it is allowed that after one coin is inserted, a coffee can be produced. It does not matter that **Implementation 1** cannot produce tea while according to the specification it is allowed. **Implementation 2** does not conform to the specification. According to the specification it is not allowed that after one coin is inserted, an espresso is produced.

Example 3.2. *Figure 3.2 depicts three more implementations of a coffee machine.*

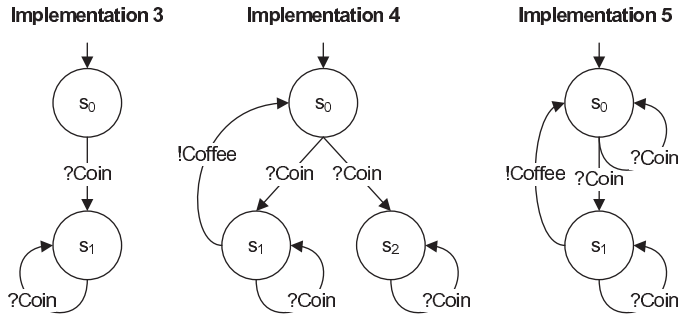

Figure 3.2: Examples where quiescence matters

Implementation 3 *is a coffee machine that accepts coins but never produces coffee.* **Implementation 4** *is a coffee machine that after receiving a coin may produce a coffee, but it can reach a state in which no coffee can be produced anymore, e.g. because it ran out of cups.* **Implementation 5** *is a coffee machine that after receiving*

a coin may produce a coffee, but sometimes multiple coins need to be inserted before a coffee can be produced, e.g. because the first coin got stuck inside the machine.

These three machines clearly should not be considered correct with respect to the specification. In these implementations a state is reached in which no output is allowed while according to the specification an output has to be allowed. However, according to the informal notion of conformance as described above these three implementations conform to the specification of Figure 3.1.

For this reason, Tretmans introduced the notion of quiescence. A quiescent state is a state in which no output action or internal action is possible and is never possible without providing input first.

Definition 3.3. *Let $\mathcal{L} = (S, s_0, A_\tau, \rightarrow)$ be an LTS, then a state $s \in S$ is quiescent, denoted by $\delta(s)$, if for all $\mu \in A_O \cup \{\tau\}$: $s \overset{\mu}{\not\rightarrow}$.*
From now on A_δ denotes a set of actions A including quiescence: $A_\delta = A \cup \{\delta\}$.

Taking into account quiescence as an output, the three implementations in Figure 3.2 do not conform to the specification of Figure 3.1. In Implementation 3, the state s_1 is a quiescent state. Therefore, after a coin is inserted quiescence is produced, which is not a subset of coffee and tea as prescribed by the specification. In Implementation 4, the state s_2 is a quiescent state. After a coin is inserted either quiescence or coffee are produced, which is also not a subset of coffee and tea as prescribed by the specification. In Implementation 5, the state s_0 is a quiescent state. After a coin is inserted either quiescence or coffee are produced, which again is not a subset of coffee and tea as prescribed by the specification.

Quiescence is viewed as observable output behavior of the implementation and the specification. Originally, Tretmans explicitly added quiescence to the specification and the implementation. Alternatively, quiescence can be added in the traces. Frantzen, Willemse and Tretmans chose this approach in [17].

A trace with quiescence labels added is called a *suspension trace*. In order to define the set of suspension traces, a new generalized transition relation is defined.

Definition 3.4. *Let $\mathcal{L} = (S, s_0, L \cup \{\tau\}, \rightarrow)$ be an LTS and let $s, s', s'' \in S$ be states. The* generalized transition relation with quiescence *is the least relation $\Rightarrow_\delta \subseteq S \times (L \cup \{\delta\})^* \times S$ such that:*

- *for all $\alpha \in L^*$, if $s \overset{\alpha}{\Rightarrow} s'$, then $s \overset{\alpha}{\Rightarrow}_\delta s'$;*
- *if $\delta(s)$, then $s \overset{\delta}{\Rightarrow}_\delta s$; and*
- *for all $\alpha, \beta \in (L \cup \{\delta\})^*$,if $s \overset{\alpha}{\Rightarrow}_\delta s'$ and $s' \overset{\beta}{\Rightarrow}_\delta s''$, then $s \overset{\alpha\beta}{\Rightarrow}_\delta s''$.*

From now on, $s \overset{\alpha}{\Rightarrow}_\delta$ denotes that there is a state s' such that $s \overset{\alpha}{\Rightarrow}_\delta s'$.

Definition 3.5. *Let $\mathcal{L} = (S, s_0, A_\tau, \rightarrow)$ be an LTS. The set of suspension traces including quiescence, denoted by $Straces(\mathcal{L})$ is defined by: $Straces(\mathcal{L}) = \{\alpha | s_0 \overset{\alpha}{\Rightarrow}_\delta\}$*

For the conformance relation, we can now define the set of output actions allowed after a sequence of labels, including quiescence as an output.

Definition 3.6. *Let $\mathcal{L} = (S, s_0, A_\tau, \rightarrow)$ be an LTS and let $s \in S$ be a state. Then, the set of states that is reachable by a sequence of labels $\alpha \in A \cup \cup\{\delta\}^*$, denoted by s* **after** *α, is defined as:*

$$s \textbf{ after } \alpha = \{s' | s \stackrel{\alpha}{\Rightarrow}_\delta s'\}.$$

For $C \subseteq S$, we write C **after** *α for $\bigcup_{s \in C} s$* **after** *α.*

Definition 3.7. *Let $\mathcal{L} = (S, s_0, A_\tau, \rightarrow)$ be an LTS and let $s \in S$ be a state. Then, the set of output actions including quiescence allowed by s, denoted by* $\mathbf{out}(s)$*, is defined as:*

$$\mathbf{out}(s) = \begin{cases} \{\delta\} & \text{if } \delta(s); \text{ and} \\ \{o \in A_O | s \stackrel{o}{\rightarrow}\} & \text{otherwise.} \end{cases}$$

For $C \subseteq S$, we write $\mathbf{out}(C)$ *for* $\bigcup_{s \in C}$ $\mathbf{out}(s)$.

These definitions make it possible to formalize the notion of conformance between an implementation and a specification.

Definition 3.8. *Let $\mathcal{I} = (I, i_0, A_\tau, \rightarrow_i)$ be an input-enabled LTS and let $\mathcal{S} = (S, s_0, A_\tau, \rightarrow_s)$ be an LTS. Then, $\mathcal{I}$ is* input output conform *$\mathcal{S}$, denoted by $\mathcal{I}$* **ioco** *$\mathcal{S}$, iff for all $\alpha \in Straces(\mathcal{S})$:*

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \alpha).$$

A test describes which input actions should be applied to the implementation and which output actions may be observed from the implementation. An output action leads to a verdict **fail** if the output action was not allowed according to the specification. If no output action is observed, while according to the specification it should be observed, then observing quiescence δ leads to the verdict **fail**. As long as the verdict **fail** is not given, a test can terminate with the verdict **pass**.

According to Tretmans, a test $\mathcal{TC} = (T \cup \{\textbf{pass}, \textbf{fail}\}, t_0, A_\delta, \rightarrow_{\mathcal{TC}})$ is a labeled transition system with a tree-like structure and **pass** or **fail** verdicts as leaves. A test is deterministic. A test has finite behavior, which means it does not allow infinite sequences of transitions. The states **pass** and **fail** are also called terminal states. These are states with no outgoing transitions. That is, for all $\mu \in A_\delta$, **pass** $\stackrel{\mu}{\not\rightarrow}_{\mathcal{TC}}$ and **fail** $\stackrel{\mu}{\not\rightarrow}_{\mathcal{TC}}$.

Example 3.9. *Figure 3.3 shows two tests derived from the specification of Figure 3.1.* Test 1 *consists of inserting a coin and concluding the verdict* ***pass*** *if a coffee or a tea is produced. If no coffee or tea is produced, which means that quiescence is observed, then the tests leads to the verdict* ***fail****.* Test 2 *is a longer test that shows that if after inserting a coin an acceptable output is observed, then the test can continue. In this test, after a coffee is produced the test continues. Since according to the specification, before a new coin is inserted no coffee or tea should be produced, the only output allowed before inserting a coin is quiescence. After quiescence is observed, a new coin can be inserted and if coffee or tea is produced Test 2 concludes the verdict* ***pass****.*

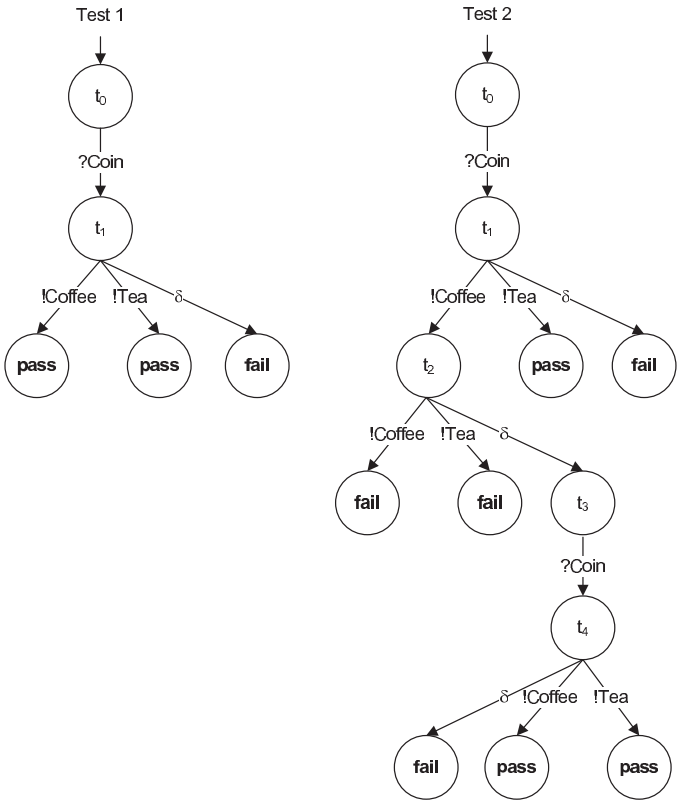

Figure 3.3: Example Test-cases

The set of all tests that can be derived from a specification is defined inductively. Note that in Definition 3.10 **pass** denotes the LTS $(\{\mathbf{pass}\}, \mathbf{pass}, \emptyset, \emptyset)$ and **fail** denotes the LTS $(\{\mathbf{fail}\}, \mathbf{fail}, \emptyset, \emptyset)$.

Definition 3.10. *Let $\mathcal{S} = (S, s_0, A_\tau, \rightarrow)$ be a specification. Let $C \subseteq S$ be a non-empty set of states of $\mathcal{S}$; then the largest set of tests that can be derived from the specification $\mathcal{S}$ starting from the set of states C, denoted by $Tests(C)$, is inductively defined as:*

1. **pass** *is an element of $Tests(C)$;*
2. *if $i \in A_I$ and C **after** $i \neq \emptyset$ and $\mathcal{TC}' \in Tests(C$ **after** $i)$, then $i;\mathcal{TC}'$ is an element of $Tests(C)$; and*
3. *if, for all $\mu \in \mathbf{out}(C)$, $\mathcal{TC}_\mu \in Tests(C$ **after** $\mu)$, then*

$$\sum\{\mu;\mathcal{TC}_\mu | \mu \in \mathbf{out}(C)\} + \\ \sum\{\mu;\mathbf{fail} | \mu \in (A_O \cup \{\delta\}) \setminus \mathbf{out}(C)\}$$

is an element of $Tests(C)$.

From now on $Tests(\mathcal{S})$ denotes the set of all tests that can be derived from $\mathcal{S}$ starting from the initial state: $Tests(\mathcal{S}) = Tests(\{s_0 \textbf{ after } \epsilon\})$.

Test execution is defined by the synchronous composition of an implementation and a test. In our definition of tests, a test contains quiescence transitions. An implementation does not contain quiescence transitions. Therefore, we need a new definition of the synchronous composition of a test including quiescence transitions, and an implementation without quiescence transitions.

Definition 3.11. *Let* $\mathcal{TC} = (T, t_0, A_\delta, \rightarrow_{\mathcal{TC}})$ *be a test case and* $\mathcal{I} = (S, s_0, A_\tau, \rightarrow)$ *be an implementation, then the* synchronous composition of $\mathcal{TC}$ and $\mathcal{I}$, *denoted by* $\mathcal{TC} \parallel_\delta \mathcal{I}$, *is defined by:*

$$\mathcal{TC} \parallel_\delta \mathcal{I} = (T \times S, (t_0, s_0), A_{\tau\delta}, \rightarrow'),$$

where

$$\begin{aligned}\rightarrow' = \{&(t,s), \mu, (t',s'))|\exists_{\mu \in A, t,t' \in T, s,s' \in S} : t \xrightarrow{\mu}_{\mathcal{TC}} t' \wedge s \xrightarrow{\mu} s'\} \cup \\ &\{((t,s), \tau, (t,s'))|\exists_{t \in T, s,s' \in S} : s \xrightarrow{\tau} s'\} \cup \\ &\{((t,s), \delta, (t',s))|\exists_{t,t' \in T, s \in S} : t \xrightarrow{\delta}_{\mathcal{TC}} t' \wedge \delta(s)\}.\end{aligned}$$

We call a suspension trace of $\mathcal{TC} \parallel_\delta \mathcal{I}$ leading to a verdict **pass** or **fail** a *test run* of $\mathcal{TC} \parallel_\delta \mathcal{I}$. An implementation $\mathcal{I}$ passes a test $\mathcal{TC}$ if all test runs lead to a verdict **pass**. An implementation passes all tests of a specification $\mathcal{S}$ if all test runs of all tests $Tests(\mathcal{S})$ lead to the verdict **pass**.

Definition 3.12. *Let* $\mathcal{TC} = (T, t_0, A_\delta, \rightarrow_t)$ *be a test and let* $\mathcal{I} = (S, s_0, A_\tau, \rightarrow)$ *be an implementation. Then,*

- *the set of all test runs of* $\mathcal{TC} \parallel_\delta \mathcal{I}$, *denoted by* $testruns(\mathcal{TC} \parallel_\delta \mathcal{I})$, *is defined as:*

$$testruns(\mathcal{TC} \parallel_\delta \mathcal{I}) = \{\alpha | \exists_{s' \in S} : (t_0, s_0) \stackrel{\alpha}{\Rightarrow}_\delta (\textbf{pass}, s') \vee (t_0, s_0) \stackrel{\alpha}{\Rightarrow}_\delta (\textbf{fail}, s')\}$$

- $\mathcal{I}$ *passes* $\mathcal{TC}$, *denoted by* $\mathcal{I}$ **passes** $\mathcal{TC}$ *iff, for all* $\alpha \in testruns(\mathcal{TC} \parallel_\delta \mathcal{I})$, *there exists an* $s' \in S$ *such that:*

$$(t_0, s_0) \stackrel{\alpha}{\Rightarrow}_\delta (\textbf{pass}, s').$$

- $\mathcal{I}$ *passes all tests of a specification* $\mathcal{S}$, *denoted by* $\mathcal{I}$ **passes** $Tests(\mathcal{S})$ *iff, for all* $\mathcal{TC} \in Tests(\mathcal{S})$: $\mathcal{I}$ **passes** $\mathcal{TC}$.

This notion of test is *sound* in the sense that a correct implementation passes all tests $Tests(\mathcal{S})$ that can be derived from the specification $\mathcal{S}$. The notion of tests is *exhaustive* in the sense that for an implementation $\mathcal{I}$ that does not conform to a specification $\mathcal{S}$, denoted by $\mathcal{I}$ **ioco** $\mathcal{S}$ there exists a test in $\mathcal{TC} \in Tests(\mathcal{S})$ that may lead to the verdict **fail**, denoted by $\mathcal{I}$ **passes** $\mathcal{TC}$.

Theorem 3.13. *If* $\mathcal{I}$ **ioco** $\mathcal{S}$ *then for all* $\mathcal{TC} \in Tests(\mathcal{S})$*:* $\mathcal{I}$ **passes** $\mathcal{TC}$ *(soundness of* ***ioco****).*

Sketch of proof. The proof is by contraposition. Let $\mathcal{I} = (S, s_0, A_\tau, \rightarrow)$ be an implementation and let $\mathcal{S}$ be a specification. Theorem 3.13 can be reformulated as: if $\mathcal{I}$ **ioco** $\mathcal{S}$ then there does not exist a $\mathcal{TC} \in Tests(\mathcal{S})$ such that there exists a suspension trace $\alpha \in testruns(\mathcal{TC} \parallel_\delta \mathcal{I})$ that leads to the verdict **fail**. Then, we suppose that $\mathcal{TC} = (T \cup \{\mathbf{pass}, \mathbf{fail}\}, t_0, A_\delta, \rightarrow_{\mathcal{TC}})$ is a test generated from $\mathcal{S}$ for which there is a test run $\alpha \in testruns(\mathcal{TC} \parallel_\delta \mathcal{I})$ and an $s' \in S$ such that $(t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{fail}, s')$. According to the definition of a test, a suspension trace can only lead to the verdict **fail** if there is a $\mu \in A_o \cup \{\delta\}$ and a suspension trace α' such that $\alpha = \alpha'\mu$ and $\mu \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha')$ and $\mu \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$. Immediately it follows that $\mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha') \not\subseteq \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$ and therefore $\mathcal{I}$ does not conform to $\mathcal{S}$ which disproves the contraposition.

Theorem 3.14. *If* $\mathcal{I}$ **io$\not$co** $\mathcal{S}$ *then there is a* $\mathcal{TC} \in Tests(\mathcal{S})$*:* $\mathcal{I}$ **pa$\not$ses** $\mathcal{TC}$ *(exhaustiveness of* ***ioco****).*

Sketch of proof. Let $\mathcal{I} = (S, s_0, A_\tau, \rightarrow)$ be an implementation and let $\mathcal{S}$ be a specification such that $\mathcal{I}$ **io$\not$co** $\mathcal{TC}$. Let $\alpha\mu$ be a test run such that $\mu \in A_O \cup \{\delta\}$ and $\mu \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha)$ and $\mu \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha)$. We inductively define a test that leads to the verdict **fail** for suspension trace $\alpha\mu$.

- Suppose $\alpha = \epsilon$ then the test

  $$\sum\{\mu; \mathbf{pass} | \mu \in \mathbf{out}(C)\} + \\ \sum\{\mu'; \mathbf{fail} | \mu' \notin (A_O \cup \{\delta\}) \setminus \mathbf{out}(C)\}\ .$$

  leads to the verdict **fail** for suspension trace $\alpha = \mu$.

- Suppose $\alpha = \nu\alpha'$, with $\nu \in A_\delta$ and $\alpha' \neq \epsilon$. Suppose there exists a test $\mathcal{TC} \in Tests(C_{\mathcal{S}}\ \mathbf{after}\ \nu)$ that leads to verdict **fail** for α' (the induction hypothesis).
  - Suppose $\alpha = i\alpha'$ then the test $i; \mathcal{TC}$ leads to the verdict **fail** for suspension trace α;
  - Suppose $\alpha = o\alpha'$, then the test

    $$o; \mathcal{TC} + \\ \sum\{\mu; \mathbf{pass} | \mu \in \mathbf{out}(C) \setminus \{o\}\} + \\ \sum\{\mu'; \mathbf{fail} | \mu' \notin (A_O \cup \{\delta\}) \setminus \mathbf{out}(C)\}\ ;$$

    leads to the verdict **fail** for suspension trace α.
  - Suppose $\alpha = \delta\alpha'$ then the test

    $$\delta; \mathcal{TC} + \\ \sum\{o; \mathbf{pass} | o \in \mathbf{out}(C) \setminus \{\delta\}\} + \\ \sum\{o'; \mathbf{fail} | o' \notin A_O \setminus \mathbf{out}(C)\}\ ;\ \textit{and}$$

    leads to the verdict **fail** for suspension trace α.

3.2 Timed Input-output Conformance and Tests

Besides input actions and output actions, a system may also have timing requirements that are to be tested. E.g. a coffee machine may have to produce a coffee within 5 seconds after a coin has been inserted. Tretmans did not define when a timed implementation is conform a timed specification and what are timed tests.

Therefore, based on Tretmans's discrete-event input-output conformance relation and notion of test, Brandán-Briones and Brinksma [12], and Krichen and Tripakis [30], and Larsen et al. [33] each developed a timed input-output conformance relation and notion of test. The relation of Larsen turned out to be the same as the relation of Krichen and Tripakis. In this section we discuss both distinctive input-output conformance relations and notions of tests are discussed. Krichen and Tripakis did not formalize their theory in the style used by Tretmans and the style used in this thesis. To show the difference between both timed notions of tests and for comparison with hybrid tests later on, we define a timed input-output conformance relation and tests in our style, closely related to the theory defined by Krichen and Tripakis.

Brandán-Briones and Brinksma [12] have defined a conformance relation and tests with a quiescence action. A state in a timed labeled transition system is quiescent if from that state no output is possible within a limited amount of time. A timed implementation is timed input-output conform a timed specification if for all reachable states, the set of output actions allowed by the implementation is a subset of all the output actions allowed by the specification. Quiescence is included in the set of outputs.

Krichen and Tripakis [30] defined a conformance relation without quiescence. In their relation, time is viewed as output of the implementation. An implementation is input-output conform a specification if, in every reachable state, the set of output actions allowed by the implementation is a subset of the output actions allowed by the specification and, in every reachable state, the time allowed to pass in the implementation is a subset of the time allowed to pass in the specification.

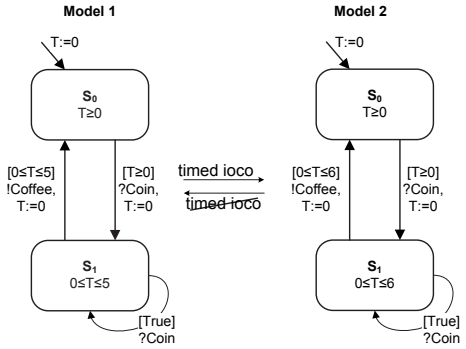

Figure 3.4: Examples of Timed Input-output Conformance

Example 3.15. *Figure 3.4 depicts two timed automata. In* **Model 1** *repeatedly a coin can be inserted after which a coffee is produced within 5 seconds. In* **Model 2** *repeatedly a coin can be inserted after which a coffee is produced within 6 seconds. Both models are considered input enabled. After a coin has been inserted another coin can be inserted, without any effect.* **Model 1** *does conform to* **Model 2** *according to both timed conformance relations. At any time that a coffee can be produced in* **Model 1**, *it can be produced in* **Model 2** *as well. However,* **Model 2** *does not conform to* **Model 1** *according to both timed conformance relations. After inserting a coin, in* **Model 2**, *a coffee can be produced after more than 5 seconds while in* **Model 1** *a coffee cannot be produced after more than 5 seconds.*

3.2.1 Timed Conformance with Quiescence

Brandán-Briones and Brinksma [12] view both the implementation and the specification as timed labeled transition systems.

In practice, quiescence cannot be observed because it cannot be observed whether an output action will never occur. Therefore, Brandán-Briones and Brinksma use a parameter M that defines the time after which quiescence can be concluded. They prove that if this M is chosen such that it is bigger than any delays before any output action of the specification has to happen, then indeed quiescence may be concluded after delay M. A state in a TLTS is M-quiescent if from that state no output actions are allowed after time M has passed.

Definition 3.16. *Let $\mathcal{T} = (S, s_0, A \cup \{\mathbb{R}^{\geq 0}\}, \rightarrow)$ be an TLTS, then a state $s \in S$ is* M-quiescent*, denoted by $\delta_M(s)$, if for all $s' \in (s$ **after** $M)$, $o \in A_O$ and $t \in \mathbb{R}^{\geq 0}$: $s' \overset{o(t)}{\not\Rightarrow}$, in which $o(t)$ denotes the sequence $t\ o$.*

From now on, for an action $\mu \in A$ and a delay $t \in \mathbb{R}^{\geq 0}$ we always use the notation $\mu(t)$ for the sequence $t\mu$.

For the definition of timed conformance Brandán-Briones and Brinksma only consider *normalized timed suspension traces*. These are suspension traces without consecutive delays. They prove that this set of traces characterizes the set of all traces in a TLTS and therefore this suffices to conclude conformance. In order to define the set of normalized timed suspension traces we need to redefine the generalized transition relation with quiescence of Tretmans [17]. (see Definition 3.4).

Definition 3.17. *Let $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a TLTS and let $s, s', s'' \in S$ be states. The* generalized transition relation with M-quiescence *is the least relation $\Rightarrow_{\delta_M} \subseteq S \times (A_\tau \cup \mathbb{R}^{\geq 0} \cup \{\delta\})^* \times S$ such that:*

- *for all $\alpha \in (A \cup \mathbb{R}^{\geq 0})^*$, if $s \overset{\alpha}{\Rightarrow} s'$, then $s \overset{\alpha}{\Rightarrow}_{\delta_M} s'$;*
- *if $\delta_M(s)$, then $s \overset{\delta(M)}{\Rightarrow}_{\delta_M} s$; and*
- *for all $\alpha, \beta \in (A \cup \mathbb{R}^{\geq 0} \cup \{\delta\})^*$,if $s \overset{\alpha}{\Rightarrow}_{\delta_M} s'$ and $s' \overset{\beta}{\Rightarrow}_{\delta_M} s''$, then $s \overset{\alpha\beta}{\Rightarrow}_{\delta_M} s''$.*

The set of normalized timed suspension traces with M-quiescence is then defined as the set of traces consisting of an alternating sequence of time labels and actions, ending with either an action label or a time label, with an action label only after a time label smaller than M and a quiescence label only after time label M. Note that in a normalized trace, a sequence of two consecutive actions a_1 and a_2 (without delay in between) is the sequence $a_1 0 a_2$.

Definition 3.18. *Let $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a TLTS and let $s \in S$ be a state. Let $\mathbb{D}$ denote the left-closed right open interval $[0, M)$; then the set of* normalized suspension traces, with quiescence only after delay M *denoted by $ntStraces_M(\mathcal{T})$, is defined as:*

$$ntStraces_M(\mathcal{T}) = \{\alpha \in (\mathbb{D}\ A + M\ \delta)^*.(\epsilon + \mathbb{D}) | s_0 \stackrel{\alpha}{\Rightarrow}_{\delta_M}\}$$

.

In order to define the timed conformance relation we redefine the set of reachable states after a trace α, using the timed generalized transition relation with M-quiescence and we define the set of allowed output actions with M-quiescence.

Definition 3.19. *Let $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a TLTS and let $s \in S$ be a state; then*

- s **after** $\alpha = \{s' | s \stackrel{\alpha}{\Rightarrow}_{\delta_M} s'\}$ *is the set of states that are reachable by trace α, from state s;*
- C **after** $\alpha = \bigcup_{s \in C} s$ **after** α, *with $C \subseteq S$.*
- $\mathbf{out}_M(s) = \{\mu(t) | \mu \in A_O \wedge 0 \leq t < M \wedge s \stackrel{\mu(t)}{\Rightarrow}\} \cup \{\delta(M) | \delta_M(s)\}$ *is the set of output actions from a state s; and*
- $\mathbf{out}_M(C) = \bigcup_{s \in C}\ \mathbf{out}_M(s)$ *with $C \subseteq S$.*

Brandán-Briones and Brinksma [12] pose some additional restrictions on the specification and the implementation. The implementation and specification are supposed to be *time divergent* and *strongly convergent*. A TLTS is time divergent if for every state there exists a trace with infinite cumulative delay. A TLTS is strongly convergent if it does not have infinite traces of internal actions. These constraints are posed to describe more realistic systems. Brandán-Briones and Brinksma also pose that the implementation has *no forced inputs*. A TLTS has no forced inputs if for every state there exists a trace without input actions and infinite cumulative delay. I.e. time is always allowed to pass without requiring an input action first.

Time divergence and strong convergence are posed to describe more realistic systems. The constraint that an implementation has no force inputs is also necessary for the correctness of the conformance relation. Otherwise, an implementation that deadlocks on an input action incorrectly conforms to a specification that does not provide this input action (but expects an output action or quiescence instead).

Definition 3.20. *Let $\mathcal{I}$ be an input-enabled TLTS. Let $\mathcal{S}$ be a TLTS. Then $\mathcal{I}$ is timed input-output conform $\mathcal{S}$ (denoted by $\mathcal{I}$ $\mathbf{tioco}_M$ $\mathcal{S}$) if and only if:*

$$\text{for all } \alpha \in ntStraces_M(\mathcal{S})\text{: } \mathbf{out}_M(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}_M(\mathcal{S} \textbf{ after } \alpha).$$

Similar to the notion of test by Tretmans, a test according to Brand'an-Briones and Brinksma is a deterministic timed labeled transition system with a tree-like structure and **pass** or **fail** verdicts as leaves. A timed test consists of input actions to be applied at a specific time, and the output actions that can be observed at every moment in time. Because in a timed test the input is selected to be applied at a selected time (in the future) it can be the case that before the input is applied, an output is observed. This has to be taken into account in a timed test. To properly take care of M-quiescence, the delay before applying an input action is chosen smaller than M and a test waits to observe an output action or quiescence for at most delay M. If after time M no output is observed, a test can wait again for time M to observe output or to conclude quiescence.

Definition 3.21. *Let $\mathcal{S} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a specification and let $C \subseteq S$ be a non-empty set of states of $\mathcal{S}$; then the set of tests that can be derived from the specification $\mathcal{S}$ starting from the set of states C, denoted by $tTests(C)$, is inductively defined as:*

1. **pass** *is an element of $tTests(C)$;*
2. *if $i \in A_I$, $0 < t < M$, and C* **after** *$i(t) \neq \emptyset$ and $\mathcal{TC}' \in Tests(C$* **after** *$i(t))$ and, for all $o \in A_O$ and $t' \leq t$ and $o(t') \in \mathbf{out}_M(C)$, $\mathcal{TC}_{o(t')} \in Tests(C$* **after** *$o(t'))$, then*

$$\begin{array}{l} t; i; \mathcal{TC}' + \\ \sum\{t'; o; \mathcal{TC}_{o(t')} | o \in A_O \wedge t' \leq t \wedge o(t') \in \mathbf{out}_M(C)\} + \\ \sum\{t'; o'; \textbf{fail} | o' \in A_O \wedge t' \leq t \wedge o(t') \notin \mathbf{out}_M(C)\} \end{array}$$

   is an element of $tTests(C)$;

3. *if, for all $o \in A_O$ and $t' < M$ and $o(t') \in \mathbf{out}_M(C)$, $\mathcal{TC}_{o(t')} \in Tests(C$* **after** *$o(t'))$, then*

   (a) *if $\delta(M) \notin \mathbf{out}_M(C)$, then*

$$\begin{array}{l} \sum\{t'; o; \mathcal{TC}_{o(t')} | o \in A_O \wedge t' < M \wedge o(t') \in \mathbf{out}_M(C)\} + \\ \sum\{t'; o'; \textbf{fail} | o' \in A_O \wedge t' < M \wedge o(t') \notin \mathbf{out}_M(C)\} + \\ M; \delta; \textbf{fail} \end{array}$$

   is an element of $tTests(C)$; and

   (b) *if $\delta(M) \in \mathbf{out}_M(C)$ and $\mathcal{TC}' \in tTests(C$* **after** *$M\delta)$, then*

$$\begin{array}{l} \sum\{t'; o; \mathcal{TC}_{o(t')} | o \in A_O \wedge t' < M \wedge o(t') \in \mathbf{out}_M(C)\} + \\ \sum\{t'; o'; \textbf{fail} | o' \in A_O \wedge t' < M \wedge o(t') \notin \mathbf{out}_M(C)\} + \\ M; \delta; \mathcal{TC}' \end{array}$$

   is an element of $tTests(C)$.

The set of timed tests that can be derived from a specification $\mathcal{S}$ from its initial state is then defined by: $tTests(\mathcal{S}) = tTests(\{s_0\})$.

Brandán-Briones and Brinksma have proven that this notion of test is sound and exhaustive with respect to the their timed input-output conformance relation. The proofs are very similar to that of soundness and exhaustiveness in the theory of Tretmans. Again, contraposition is used to prove soundness and a test is constructed from a trace that leads to verdict **fail** to prove exhaustiveness. For simplicity reasons, in both proofs normalized traces are used.

3.2.2 Timed Conformance without Quiescence

Krichen and Tripakis [30], and simultaneously Larsen et al. have define a timed input-output conformance relation without the notion of quiescence. Instead, they take the time that may elapse from the reachable states into account. If after a trace, the time that may elapse in the implementation is larger than the the time that may elapse in the specification, then the implementation is not conform the specification. If the implementation is not conform the specification for this reason, it means that the implementation could delay performing any output action while according to the specification an output action must have occurred. In this way, time is viewed as output of the implementation.

For this relation, we define the set of normalized timed traces. A normalized trace is a trace without consecutive delays. E.g. for $t_1, t_2 \in \mathbb{R}^{\geq 0}$ and $a_1, a_2 \in A$, the traces $t_1 a_1 t_2 a_2$, $t_1 a_1 a_2 t_2$ and $a_1 t_1 a_2 t_2$ are normalized traces. Because of the constraints **C1**, **C2**, and **C3**, which we posed on TLTSs, the set of normalized traces characterizes all traces. Krichen and Tripakis do not use normalized traces. We do this to simplify the soundness and exhaustivemess proofs for our formal notion of test based on the test generation procedure algorithm of Krichen and Tripakis. First, we define a generalized transition relation for normalized timed traces. In this definition, a long arrow is simply used to fit the label above the generalized transition.

Definition 3.22. *Let $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a TLTS. Let $s, s', s'' \in S$ be states. The generalized transition relation for normalized timed traces, denoted by $\Rightarrow_{nt} \subseteq S \times (A \cup \mathbb{R}^{\geq 0})^* \times S$, is defined as:*

- *if $s \stackrel{\epsilon}{\Rightarrow} s$, then $s \stackrel{\epsilon}{\Rightarrow}_{nt} s'$;*
- *for all $a \in A$, if $s \stackrel{a}{\Rightarrow} s'$, then $s \stackrel{a}{\Rightarrow}_{nt} s'$;*
- *for all $t \in \mathbb{R}^{\geq 0}$, if $s \stackrel{t}{\Rightarrow} s'$, then $s \stackrel{t}{\Rightarrow}_{nt} s'$;*
- *for all $\alpha, \beta \in (A \cup \mathbb{R}^{\geq 0})^*$ and $\mu \in A$, if $s \stackrel{\alpha\mu}{\Rightarrow}_{nt} s'$ and $s' \stackrel{\beta}{\Rightarrow}_{nt} s''$, then $s \stackrel{\alpha\mu\beta}{\Rightarrow}_{nt} s''$;*
- *for all $\alpha, \beta \in (A \cup \mathbb{R}^{\geq 0})^*$ and $\mu \in A$, if $s \stackrel{\alpha}{\Rightarrow}_{nt} s'$ and $s' \stackrel{\mu\beta}{\Rightarrow}_{nt} s''$, then $s \stackrel{\alpha\mu\beta}{\Rightarrow}_{nt} s''$; and*
- *for all $\alpha, \beta \in (A \cup \mathbb{R}^{\geq 0})^*$ and $t_1, t_2 \in \mathbb{R}^{\geq 0}$, if $s \stackrel{\alpha t_1}{\Rightarrow}_{nt} s'$ and $s' \stackrel{t_2\beta}{\Rightarrow}_{nt} s''$, then $s \stackrel{\alpha\, t_1+t_2\, \beta}{\Longrightarrow}_{nt} s''$.*

Definition 3.23. *Let $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a TLTS; then the set of* normalized timed traces *denoted by $nttraces(\mathcal{T})$, is defined as:*

$$nttraces(\mathcal{T}) = \{\alpha | s_0 \stackrel{\alpha}{\Rightarrow}_{nt}\}$$

.

In order to define timed conformance according to Krichen and Tripakis, the set of reachable states after a trace α and the set of allowed output actions is defined as follows. Note that in Definition 3.24 we use the generalized transition relation of Definition 2.2.

Definition 3.24. *Let $\mathcal{T} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a timed transition system and let $s \in S$ be a state. Let $\alpha \in (A \cup \mathbb{R}^{\geq 0})^*$. Then:*

- s **after** $\alpha = \{s' | s \stackrel{\alpha}{\Rightarrow} s'\}$ *is the set of states that are reachable by α, from state s;*
- C **after** $\alpha = \bigcup_{s \in C} s$ **after** α, *with $C \subseteq S$;*
- $\mathbf{elapse}(s) = \{t > 0 | s \stackrel{t}{\Rightarrow}\}$ *denotes the time that is allowed to pass from state s*
- $\mathbf{out}(s) = \{o | o \in A_O \wedge s \stackrel{o}{\Rightarrow}\} \cup \mathbf{elapse}(s)$ *denotes the set of all possible outputs from state s; and*
- $\mathbf{out}(C) = \bigcup_{s \in C}$ $\mathbf{out}(s)$ *with $C \subseteq S$.*

Now the set of output contains only the set of output actions that are allowed without delay, and the set of all possible delays.

Definition 3.25. *Let $\mathcal{I}$ be an input-enabled TLTS. Let $\mathcal{S}$ be a TLTS. Then $\mathcal{I}$ is timed input-output conform $\mathcal{S}$ (denoted by $\mathcal{I}$* **tioco** *$\mathcal{S}$) iff*

$$\text{for all } \alpha \in nttraces(\mathcal{S}): \mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \alpha).$$

Krichen and Tripakis only give an informal description of their tests. According to their test generation algorithm, a test is either selecting an input action (to be applied immediately), or waiting for an output action for a selected amount of time. If during the waiting period an output action is observed and the output action is allowed according to the specification, the test may continue. If during the waiting period an output action is observed that is not allowed according to the specification, then the verdict **fail** is given and testing stops. If the waiting period ends without observing any output, but the specification allows more delay, testing may continue. If according to the specification no more waiting is allowed and an output action must occur, but the implementation does not produce this output, then the verdict fail is given.

Definition 3.26. *Let $\mathcal{S} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be a specification and let $C \subseteq S$ be a non-empty set of states; then the set of timed tests that can be derived from the specification $\mathcal{S}$ from the set of states C, denoted by $tTests(C)$, is inductively defined as follows:*

1. **pass** *is an element of $tTests(C)$;*
2. *if $i \in A_I$ and C **after** $i \neq \emptyset$ and $\mathcal{TC}' \in Tests(C$ **after** $i)$, then $i;\mathcal{TC}'$ is an element of $tTests(C)$;*
3. *let $t \in \mathbb{R}^{>0}$, and for all $o \in \mathbf{out}(C), \mathcal{TC}_o \in tTests(C$ **after** $o)$; and for all $0 < t' < t$ and $o \in \mathbf{out}(C$ **after** $t')$, $\mathcal{TC}_{o(t')} \in tTests(C$ **after** t' $o)$; and $\mathcal{TC}_t \in tTests(C$ **after** $t)$, then*

   (a) *if $t \in \mathbf{out}(C)$ then*

$$\begin{array}{l}\sum\{o;\mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C \textbf{ after } o)\} + \\ \sum\{o;\textbf{fail} | o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } o)\ \}+ \\ \sum\{t';o;\mathcal{TC}_{t'o} | 0 < t' \leq t \wedge o \in A_O \cap \mathbf{out}(C \textbf{ after } t')\} + \\ \sum\{t';o;\textbf{fail} | 0 < t' \leq t \wedge o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } t')\ \}+ \\ t;\mathcal{TC}_t\end{array}$$

   ; and

   (b) *if $t \notin \mathbf{out}(C)$ then*

$$\begin{array}{l}\sum\{o;\mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C \textbf{ after } o)\} + \\ \sum\{o;\textbf{fail} | o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } o)\ \}+ \\ \sum\{t';o;\mathcal{TC}_{t'o} | t' \in \mathbf{out}(C)\ \wedge o \in A_O \cap \mathbf{out}(C \textbf{ after } t')\} + \\ \sum\{t';o;\textbf{fail} | t' \notin \mathbf{out}(C) \vee o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } t')\}+ \\ t;\textbf{fail}\end{array}$$

   is an element of $tTests(C)$.

The set of timed tests that can be derived from a specification $\mathcal{S}$ from its initial state is then defined by $tTests(\mathcal{S}) = tTests(\{s_0\})$.

This notion of test is sound and exhaustive with respect to the **tioco** relation. Because we have formalized the definition of tests according to Krichen and Tripakis, we can give proofs here. The proofs are very similar to the proofs of soundness and exhaustiveness for the conformance relation by Tretmans. Like for timed tests according to Brandán and Brinksma it is sufficient to prove soundness and exhaustiveness with test runs of normalized traces.

First we define when a timed implementation passes all timed tests that can be derived from a specification $\mathcal{S}$. Note that in Definition 3.27 we use the synchronous composition of Definition 2.7.

Definition 3.27. *Let $\mathcal{TC} = (T, t_0, A \cup \mathbb{R}^{\geq 0}, \rightarrow_t)$ be a timed test and let $\mathcal{I} = (S, s_0, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow)$ be an implementation.*

- *the set of all test runs of $\mathcal{TC} \parallel \mathcal{I}$, denoted by $testruns(\mathcal{TC} \parallel \mathcal{I})$, is defined as:*

  $$testruns(\mathcal{TC} \parallel \mathcal{I}) = \{\alpha | \exists_{s' \in S} : (t_0, s_0) \stackrel{\alpha}{\Rightarrow}_{nt} (\mathbf{pass}, s') \vee (t_0, s_0) \stackrel{\alpha}{\Rightarrow}_{nt} (\mathbf{fail}, s')\}$$

- *$\mathcal{I}$ passes $\mathcal{TC}$, denoted by $\mathcal{I}$ **passes** $\mathcal{TC}$ iff, for all $\alpha \in testruns(\mathcal{TC} \parallel \mathcal{I})$, there exists an $s' \in S$ such that:*

  $$(t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{pass}, s').$$

- *$\mathcal{I}$ passes all tests of a specification $\mathcal{S}$, denoted by $\mathcal{I}$ **passes** $tTests(\mathcal{S})$ iff, for all $\mathcal{TC} \in tTests(\mathcal{S})$: $\mathcal{I}$ **passes** $\mathcal{TC}$.*

Theorem 3.28. *Let $\mathcal{I} = (S_\mathcal{I}, s_{0\mathcal{I}}, A_\tau \cup \mathbb{R}^{\geq 0}, \rightarrow_\mathcal{I})$ be an implementation and let $\mathcal{S}$ be a specification. If $\mathcal{I}$ **tioco** $\mathcal{S}$ then for all $\mathcal{TC} \in tTests(\mathcal{S})$: $\mathcal{I}$ **passes** $\mathcal{TC}$ (soundness of **tioco**).*

Sketch of proof. The proof is by contraposition. Suppose that there is a $\mathcal{TC}$ generated from $\mathcal{S}$ that is not sound. Then there is a test run α and state $s \in S_\mathcal{I}$ such that $(t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{fail}, s)$. According to the definition of a test, a test run leads to the verdict **fail** if there is a $\mu \in A_o \cup \mathbb{R}^{>0}$ and a α' such that $\alpha = \alpha'\mu$ and $\mu \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha')$ and $\mu \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$. However. $\mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha') \subseteq \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$ because $\mathcal{I}$ **ioco** $\mathcal{S}$, and therefore $\mu \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha') \implies \mu \in \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$.

Theorem 3.29. *If $\mathcal{I}$ **ioco** $\mathcal{S}$ then there is a $\mathcal{TC} \in tTests(\mathcal{S})$: $\mathcal{I}$ **passes** $\mathcal{TC}$ (exhaustiveness of **tioco**).*

Sketch of proof. The proof is given by constructing a $\mathcal{TC}$ from $\mathcal{S}$ that may lead to the verdict **fail**. Let $\alpha\mu$ be a normalized timed trace such that $\mu \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$ and $\mu \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha')$. Because $\mathcal{I}$ **ioco** $\mathcal{S}$ we know such trace α exists. We inductively define a test that leads to the verdict **fail** for trace $\alpha = \alpha'\mu$.

- Suppose that $\alpha' = \epsilon$.
  - Suppose $\mu = o$, then, for all $t \in \mathbb{R}^{>0}$ and $0 < t' < t$,
    - if $t \in \mathbf{out}(C)$ then the test

      $$\begin{array}{l} \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C\ \mathbf{after}\ o)\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \wedge o \notin \mathbf{out}(C\ \mathbf{after}\ o)\ \} + \\ \sum\{t'; o; \mathbf{pass} | 0 < t' \leq t \wedge o \in A_O \cap \mathbf{out}(C\ \mathbf{after}\ t')\} + \\ \sum\{t'; o; \mathbf{fail} | 0 < t' \leq t \wedge o \in A_O \wedge o \notin \mathbf{out}(C\ \mathbf{after}\ t')\ \} + \\ t; \mathbf{pass} \end{array}$$

      ; or
    - if $t \notin \mathbf{out}(C)$ then the test

      $$\begin{array}{l} \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C\ \mathbf{after}\ o)\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \wedge o \notin \mathbf{out}(C\ \mathbf{after}\ o)\ \} + \\ \sum\{t'; o; \mathbf{pass} | t' \in \mathbf{out}(C)\ \wedge o \in A_O \cap \mathbf{out}(C\ \mathbf{after}\ t')\} + \\ \sum\{t'; o; \mathbf{fail} | t' \notin \mathbf{out}(C) \vee o \in A_O \wedge o \notin \mathbf{out}(C\ \mathbf{after}\ t')\} + \\ t; \mathbf{fail} \end{array}$$

leads to the verdict **fail** for trace α.

– Suppose $\mu = t$ then, for all $0 < t' < t$ the test

$$\begin{array}{l}\sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C \textbf{ after } o)\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } o)\ \}+ \\ \sum\{t'; o; \mathbf{pass} | t' \in \mathbf{out}(C)\ \wedge o \in A_O \cap \mathbf{out}(C \textbf{ after } t')\} + \\ \sum\{t'; o; \mathbf{fail} | t' \notin \mathbf{out}(C) \vee o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } t')\}+ \\ t; \mathbf{fail}\end{array}$$

leads to the verdict **fail** for trace α.

- Suppose $\alpha = \nu\alpha''$, with $\nu \in A \cup \mathbb{R}^{\geq 0}$ and $\alpha'' \neq \epsilon$. Suppose there exists a test $\mathcal{TC} \in tTests(C_{\mathcal{S}} \textbf{ after } \nu)$ that leads to the verdict **fail** for α'' (the induction hypothesis).

  – Suppose $\alpha = i\alpha''$ then the test $i; \mathcal{TC}$ leads to the verdict **fail** for trace α.

  – Suppose $\alpha = o\alpha''$, then for all $t \in \mathbb{R}^{>0}$ and $0 < t' < t$,

    * if $t \in \mathbf{out}(C)$ then the test

$$\begin{array}{l}o; \mathcal{TC}+ \\ \sum\{o'; \mathbf{pass} | o' \in (A_O \cap \mathbf{out}(C \textbf{ after } o)) \setminus \{o\}\} + \\ \sum\{o'; \mathbf{fail} | o' \in A_O \wedge o' \notin \mathbf{out}(C \textbf{ after } o)\ \}+ \\ \sum\{t'; o'; \mathbf{pass} | 0 < t' \leq t \wedge o' \in A_O \cap \mathbf{out}(C \textbf{ after } t')\} + \\ \sum\{t'; o'; \mathbf{fail} | 0 < t' \leq t \wedge o' \in A_O \wedge o' \notin \mathbf{out}(C \textbf{ after } t')\ \}+ \\ t; \mathbf{pass}\end{array}$$

    ; or

    * if $t \notin \mathbf{out}(C)$ then the test

$$\begin{array}{l}o; \mathcal{TC}+ \\ \sum\{o'; \mathbf{pass} | o' \in (A_O \cap \mathbf{out}(C \textbf{ after } o)) \setminus o\} + \\ \sum\{o'; \mathbf{fail} | o' \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } o)\}+ \\ \sum\{t'; o'; \mathbf{pass} | t' \in \mathbf{out}(C)\ \wedge o' \in A_O \cap \mathbf{out}(C \textbf{ after } t')\} + \\ \sum\{t'; o'; \mathbf{fail} | t' \notin \mathbf{out}(C) \vee o' \in A_O \wedge o' \notin \mathbf{out}(C \textbf{ after } t')\}+ \\ t; \mathbf{fail}\end{array}$$

    leads to the verdict **fail** for trace α.

  – Suppose $\alpha = t\alpha''$, then the test

$$\begin{array}{l}\sum\{o; \mathbf{pass} | o \in (A_O \cap \mathbf{out}(C \textbf{ after } o)) \setminus \{o\}\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } o)\ \}+ \\ \sum\{t'; o; \mathbf{pass} | 0 < t' \leq t \wedge o \in A_O \cap \mathbf{out}(C \textbf{ after } t')\} + \\ \sum\{t'; o; \mathbf{fail} | 0 < t' \leq t \wedge o \in A_O \wedge o \notin \mathbf{out}(C \textbf{ after } t')\ \}+ \\ t; \mathcal{TC}\end{array}$$

  leads to the verdict **fail** for trace α.

3.2.3 Comparison

It is possible that an implementation conforms to a specification according to the conformance relation by Krichen and Tripakis, but that the same implementation does not conform to the same specification according to the conformance realation by Brandán-Briones and Brinksma.

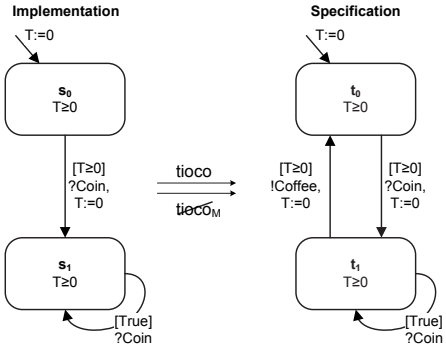

Figure 3.5: Difference between timed conformance relations

Example 3.30. *The implementation in Figure 3.5 accepts a coin, after which no output actions can occur. However, it can still delay and receive input actions. The specification repetitively accepts a coin and after that may produce a coffee. In this case the implementation does not conform to the specification according to Brandán-Briones and Brinksma because in location* s_1 *the system is quiescent, while it is not quiescent in the specification and therefore:* $\mathbf{out}_M(\{s_0\}\ \mathbf{after}\ ?Coin) \not\subseteq \mathbf{out}_M(\{\ t_1\}\ \mathbf{after}\ ?Coin)$ *because* $\{\delta(M)\} \not\subseteq \{!Coffee\}$. *However, the implementation does conform to the specification according to Krichen and Tripakis. In this case* $\mathbf{out}(\{s_0\}\ \mathbf{after}\ ?Coin) \subseteq \mathbf{out}(\{t_0\}\ \mathbf{after}\ ?Coin)$ *because* $\{t|t \in \mathbb{R}^{\geq 0}\} \subseteq \{!Coffee\} \cup \{t|t \in \mathbb{R}^{\geq 0}\}$.

Conversely, Krichen and Tripakis [32] have proven that if an implementation conforms to a specification according to Brandán-Briones and Brinksma, then the implementation also conforms to the specification according to Krichen and Tripakis. Schmaltz and Tretmans [43] have proven that if the implementation does not contain quiescent states, then the two timed conformance relations are the same.

3.3 Concluding Remarks

In this section we described several input-output conformance relations and their notions of tests. This chapter shows that different choices can be made in defining a conformance relation and a notion of test. The use of quiescence is convenient, because it allows for a stricter conformance in the sense that more implementations can be distinguished from each other. However, in practice it cannot be implemented.

The reason is that in practice it is impossible to conclude whether no output action will ever take place. Therefore, both timed input-output conformance relations and notions of test have validity.

In the next chapter we define a hybrid input-output conformance relation that takes into account both discrete-event and continuous behavior of an implementation. We will see that the conformance relation becomes more complex compared to the other conformance relations because continuous input and continuous output take place synchronously.

Like with the other conformance theories we have to decide whether we use a similar notion to quiescence. Continuous output always takes place. Quiescence, meaning no output will ever take place without providing input first, does not apply. We will see that for hybrid systems we need a notion of agile state, indicating that the output actions do not have to take place immediately.

4

Hybrid Input-output Conformance and Tests

This chapter describes a hybrid conformance relation and a notion of hybrid tests. In order to get some feeling on how tests should be derived from a hybrid specification we start with an example. After that, we formally define hybrid conformance and the notion of hybrid test, and we prove that our notion of hybrid test is sound and exhaustive with respect to the conformance relation.

The hybrid conformance relation and notion of hybrid test presented in this chapter were first published as [39]. An extended version of that paper, including the soundness and exhaustiveness proofs was published as [40].

4.1 An Informal Introduction to Hybrid Testing

We illustrate testing of a hybrid system with continuous input by means of an example. We consider a thermostat plus its environment.

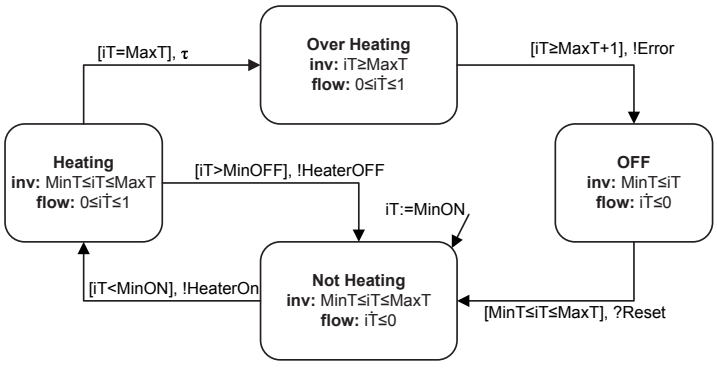

Figure 4.1: Thermostat Example

Example 4.1. *Figure 4.1 shows the hybrid automaton of the thermostat plus the temperature input and an input action it may receive from a chamber. The thermo-*

stat in this example receives continuous input, namely temperature iT, it receives input actions, namely !Reset, and it produces output actions, namely !HeaterOn and !HeaterOFF. $MinT$, $MaxT$, $MinON$, and $MinOFF$ are constant values. We assume that $MinT < MinON < MinOFF < MaxT$. The thermostat maintains the temperature of a room between and including the temperatures $MinT$ and $MaxT$. It does this by switching on a heater if the temperature drops below $MinON$ and it switches off a heater if the temperature rises above $MinOFF$.

Initially the chamber temperature iT satisfies $MinT \leq T \leq MaxT$ and the thermostat is in the mode **Not Heating**. *The rate of change of iT is not positive (the temperature stays the same or decreases). The thermostat can turn on a heater if the temperature is below the specified temperature MinON. The thermostat in this case switches to the mode* **Heating** *in which the chamber is being heated with a maximum rate between and including* 0 *and* 1. *After the temperature MinOFF is reached the thermostat returns to the mode* **Heater OFF** *with the message !HeaterOFF and the temperature in the chamber starts to decrease (e.g. because the chamber is not perfectly isolated and the temperature is colder outside the chamber). If the temperature iT reaches $MaxT$, then the thermostat switches to the mode* **Over Heating**. *This is accompanied by an internal action τ. In mode* **Over Heating** *the temperature still increases, but before the temperature increases with more than 1 ^oC the thermostat switches to mode* **OFF** *and produces the message !Error. After the error occurred, the thermostat can only be reset by a discrete input action ?Reset (e.g. a button being pressed by an operator).*

Note that this specification does not accept every possible input because the temperature always increases or decreases with a maximum of 1 $^oC/min$. Tests generated from this specification will not contain the input behavior of increasing or decreasing temperature with more than 1 $^oC/min$.

Following the line of the notions of test presented in Chapter 3, a test for a hybrid system will consist of steps of the following types:

- select an input action from the set of input actions possible according to the specification and apply it;
- try to observe an output action from the implementation; if an output action is observed which is not allowed according to the specification, then terminate with a verdict fail; if no output action is observed when it is supposed to be observed, then terminate with a verdict fail as well; or
- select continuous input for a specific duration and apply it; simultaneously observe continuous output for the selected duration or until an output action is observed; if the observed output is not allowed according to the specification, then the terminate with a verdict fail.

As long as the verdict fail is not concluded, the test continues with another step or terminates with a verdict pass.

Example 4.2. *Let in the above example $MinT = 5$ ^oC, let $MinOn = 10$ ^oC, let $MinOFF = 15$ ^oC, and let $MaxT = 20$ ^oC. Let the initial temperature be $T =$*

MinOn. Then, the following sequence of steps describes a test scenario for the thermostat.

step 1: *Decrease the temperature with* 0.5 $^oC/min$ *for one minute. If a !HeaterON output or no output is observed, then continue testing with step 2, otherwise (if e.g. a !heaterOFF output or an error output is observed which is not allowed according to the specification) or if no output is observed stop testing and conclude with verdict fail.*

The implementation can either have produced no output, the !heaterON output, the !HeaterOFF output, or the !Error output. Suppose that the implementation produced the !heaterON output after one minute, then immediately after the output is observed and validated a new step has to be performed.

step 2: *Select a temperature increase with* 1 $^oC/min$ *for ten minutes.*

At the start of step 2 the temperature iT is 9.5 ^oC*. Therefore, at the end of step 2 the temperature iT is* 19.5 ^oC*. The test can continue as follows.*

step 3: *Select a temperature increase of* 0.5 $^oC/min$ *for three minutes. . During this temperature increase, the thermostat went to mode* **Over Heating** *and the end temperature is* 21 ^oC*. Therefore, if the output action !Error is observed, then the test can continue with a new step or stop with the verdict pass, otherwise (if no output !Error is observed, or an output action !HeaterON or !HeaterOFF is observed) stop testing with the verdict fail.*

Suppose that the output !Error was observed from the implementation. Then it is possible to continue the test by applying a decreasing temperature at most until the temperature reaches 5 ^oC *and applying an input action ?Reset. It is also possible to stop testing at this point with verdict pass, because in this case the test did not fail.*

It is also possible to define tests for hybrid systems with continuous output. In this case, we need to compare the continuous output of the implementation with the specified continuous output. An example of such a system is a robot arm which is required to move according to a specific speed. It is also possible to define tests for hybrid systems with both continuous input and continuous output. In this case, we need to compare the continuous output of the implementation with the specified continuous output synchronously with applying continuous input to the specification. An example of such a system is a brake control system of a car that, besides displaying discrete event behavior, brakes in accordance with the pressure applied to the brake pedal. It is also possible to describe tests for hybrid systems with multiple continuous input and multiple continuous output. An example of a system with multiple continuous input is a vacuum system controller which observers both the pressure in the chamber and the pressure in a pipe connected to a pump (see Chapter 7). For all these kinds of hybrid systems we define a conformance relation and a notion of test.

4.2 Specification and Implementation

Before we define hybrid input-output conformance we first have to make some decisions on how we interpret a specification and implementation. E.g. we have to decide in case both a trajectory is allowed and an output action is allowed, whether, if no continuous input is applied to the implementation, whether the implementation is considered to perform the output action or not.

We distinguish actions that must happen from actions that may happen as follows. Consider the HTS fragments in Figure 4.2.

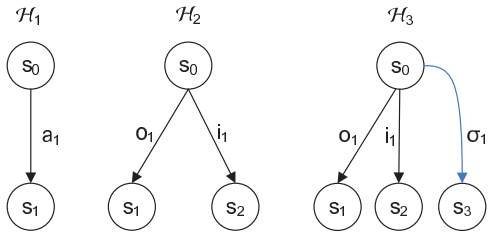

Figure 4.2: HTS Interpretation

$\mathcal{H}_1$ models that the action a_1 has to be performed. $\mathcal{H}_2$ models that either input action i_1 has to be performed or output action o_1 has to be performed without delay. $\mathcal{H}_2$ could be a fragment of an implementation and models that if an input action is not applied, then the output action is performed without delay. $\mathcal{H}_3$ models that an output action is allowed, but it does not have to take place. This fragment may occur in both the specification and in the implementation (if the state s_0 besides these transitions allows any input behavior).

Furthermore, we have to decide whether we need additional constraints on our specification. Zeno behavior is the behavior of an infinite sequence of transitions whose accumulative duration is finite. We do do not specifically disallow such behavior. However, in practice, an implementation never displays Zeno behavior because time will always progresses. In our theory, Zeno behavior is not an issue because we consider finite traces only.

Brandán-Briones and Brinksma imposed three additional constraints on their TLTS (see Section 3.2.1): time divergence, strong convergence, and no forced input. This raises the question whether we need time divergence, strong convergence, and no forced input (for both input actions and continuous input). Since time divergence is only a practical issue, i.e. in practice an implementation is always time divergent because time will always progress, we do not impose it on the implementation or the specification. Since we only consider finite traces, strong convergence is not an issue either. No forced inputs, for us, is also only a practical issue since it makes the progress of time stop, and therefore an implementation cannot have forced input.

4.3 Hybrid Input-output Conformance

In this section we define the conformance relation for hybrid systems. We first define the conformance relation for hybrid systems without continuous input behavior. The definition of this notion is fairly straightforward. We present it for didactical reasons; it allows us to emphasize the complications introduced by continuous input. Then, we define a conformance relation that includes continuous input behavior. This notion is more complex because continuous input and continuous output take place synchronously. This second conformance relation also decides conformance between HTSs without continuous input or HTSs without continuous output.

We also define a conformance relation in which the specification consists of a model of the implementation and a model of the environment. The environment specifies the surroundings of the implementation. The environment describes the input for the implementation with respect to the output performed by the implementation. It specifies the surroundings of the implementation under test, e.g. if the implementation (and specification) is a thermostat, then the environment specifies the room with heater in which it is placed. Using an explicit environment simplifies the definition of the conformance relation. We prove that the various notions of conformance coincide.

4.3.1 Hybrid Conformance with Continuous Output Only

For systems with input actions, output actions, and with continuous output only, we consider a hybrid implementation conform to a hybrid specification if, in every reachable state, the output trajectories and output actions allowed by the implementation form a subset of the output trajectories and output actions allowed by the specification. This relation is based on the discrete-event conformance relation by Tretmans [48] with continuous output behavior. Unlike Tretmans's relation quiescence does not need to be taken into account since continuous output always takes place. This relation for hybrid systems is similar to the timed relation of Krichen and Tripakis [30]. If we view time as continuous output of the implementation, then instead of taking the time that is allowed to elapse into account in the conformance relation, we now take all continuous output that is allowed to take place into account in the conformance relation.

First, we define the set of states reachable from a state or from a set of states, and we define the set of output actions allowed in a state or in a set of states, without quiescence.

Definition 4.3. *Let $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a HTS and let $\alpha \in (A \cup \Sigma)^*$. For a state $s \in S$ we define:*

$$s \textbf{ after } \alpha = \{s' | s \stackrel{\alpha}{\Rightarrow} s'\}.$$

For a set of states $C \subseteq S$ we define:

$$C \textbf{ after } \alpha = \bigcup_{s \in C} s \textbf{ after } \alpha.$$

Definition 4.4. *Let $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a HTS. For a state $s \in S$ we define:*

$$\mathbf{out}(s) = \{o \in A_O | s \stackrel{o}{\Rightarrow}\}.$$

For states $C \subseteq S$ we define:

$$\mathbf{out}(C) = \bigcup_{s \in C} \mathbf{out}(s).$$

Note that we use the generalized transition relation of Definition 2.2.

Then, we define the set of trajectories allowed in a state and the set of trajectories allowed in a set of states.

Definition 4.5. *Let $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a HTS and let $s \in S$ be a state of $\mathcal{H}$; then:*

$$\mathbf{traj}(s) = \{\sigma \in \Sigma | s \stackrel{\sigma}{\Rightarrow}\}.$$

For a set of states $C \subseteq S$ we define:

$$\mathbf{traj}(C) = \bigcup_{c \in C} \mathbf{traj}(c).$$

Let $\mathcal{I}$ be an input-enabled hybrid implementation and let $\mathcal{S}$ be a hybrid specification, both with a set of trajectories Σ on a set of output variables. Then $\mathcal{I}$ conforms to $\mathcal{S}$ if and only if for al traces $\alpha \in traces(\mathcal{S})$:

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \alpha) \wedge$$

$$\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha).$$

Example 4.6. *Consider the fragments of two hybrid transition systems shown in Figure 4.3.*

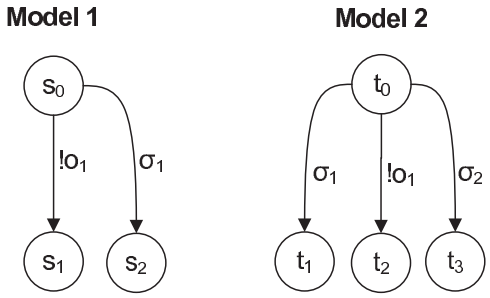

Figure 4.3: Two Model Fragments

The fragment of **Model 1** *shows the state s_0 of* **Model 1**, *which allows the output action o_1 and the trajectory σ_1. The fragment of* **Model 2** *shows the state t_0 of* **Model 2**, *which allows the output action o_1 and the trajectories σ_1 and σ_2 (with $\sigma_1 \neq \sigma_2$). For state s_0 of* **Model 1** *and state t_0 of* **Model 2** *it holds that* $\mathbf{out}(t_0) = \mathbf{out}(s_0)$ *and* $\mathbf{traj}(s_0) \subseteq \mathbf{traj}(t_0)$. *Therefore, the fragment of* **Model 1** *conforms to the fragment of* **Model 2**. *However,* **Model 2** *does not conform to* **Model 1** *because in state t_0 $\{\sigma_1, \sigma_2\} \not\subseteq \{\sigma_1\}$.*

4.3.2 Conformance with Continuous Input and Continuous Output

We now consider hybrid systems with input actions, output actions, continuous output and continuous input. Still, the conformance relations described in this section also apply for hybrid systems without continuous input as well. Like for discrete-event conformance and timed conformance, we assume that the implementation is input enabled. This means that at every moment in time, every possible continuous input is allowed. The specification on the other hand is not required to be input enabled; thus it is possible to test an implementation with respect to the input provided by the specification. As a result, the conformance relation explained in the previous section is not useful anymore because (in general), if continuous input is involved, the set of trajectories allowed by the implementation is not a subset of the set of trajectories allowed by the specification, after any trace. For the previously explained conformance relation, an input enabled implementation never conforms to a specification that is not input enabled.

Example 4.7. *Let, in Figure 4.3, σ_1 and σ_2 be trajectories on a non-empty set of output variables V_O and a a non empty set of variables V_I. Let the fragment of* **Model 2** *be input enabled. If $\sigma_1 \downarrow V_I \neq \sigma_2 \downarrow V_I$ then* $\mathbf{traj}(t_0) \not\subseteq \mathbf{traj}(s_0)$ *and therefore the fragment of* **Model 2** *does not conform to the fragment of* **Model 1**. *However, we want to take into account that if the fragment of* **Model 1** *is part of the specification and the input of σ_1 is applied to the fragment of* **Model 2**, *then σ_2 is never performed and therefore the fragment of* **Model 2** *should conform to the fragment of* **Model 1**.

A conformance relation should only take into consideration the output of the implementation that may arise if input is applied according to the specification. To achieve this, we filter the set of trajectories allowed by the implementation after a trace, with respect to the continuous input allowed by the specification after that trace.

Definition 4.8. *Let $\Sigma_{\mathcal{I}}$ and $\Sigma_{\mathcal{S}}$ be two sets of trajectories on a set of variables V with input variables $V_I \subseteq V$; then:*

$$\mathbf{infilter}(\Sigma_{\mathcal{I}}, \Sigma_{\mathcal{S}}) = \{\sigma \in \Sigma_{\mathcal{I}} | \exists_{\sigma' \in \Sigma_{\mathcal{S}}} : \sigma \downarrow V_I = \sigma' \downarrow V_I\}.$$

Let us consider the conformance relation that is obtained by adapting the conformance relation of Section 4.3.1 using the notion of filtering: an input-enabled hybrid

implementation $\mathcal{I}$ conforms to a hybrid specification $\mathcal{S}$ if and only if for all traces $\alpha \in traces(\mathcal{S})$:

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \alpha) \wedge$$

$$\mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha), \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha)) \subseteq \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha).$$

Example 4.9. *Let, in Figure 4.3, σ_1 and σ_2 be trajectories on a non-empty set of output variables V_O and a a non empty set of variables V_I. If $\sigma_1 \downarrow V_I \neq \sigma_2 \downarrow V_I$ then*

$$\mathbf{infilter}(\mathbf{traj}(t_0), \mathbf{traj}(s_0)) = \mathbf{infilter}(\{\sigma_1, \sigma_2\}, \{\sigma_1\}) = \{\sigma_1\}.$$

This means that, since $\mathbf{out}(t_0) \subseteq \mathbf{out}(s_0)$, the fragment of **Model 2** *conforms to the fragment of* **Model 1**. *If $\sigma_1 \downarrow V_I = \sigma_2 \downarrow V_I$, then* **Model 2** *does not conform to* **Model 1** *because then $\mathbf{infilter}(\{\sigma_1, \sigma_2\}, \{\sigma_1\}) = \{\sigma_1, \sigma_2\}$ and therefore $\mathbf{infilter}(\mathbf{traj}(t_0), \mathbf{traj}(s_0)) \not\subseteq \mathbf{traj}(s_0)$.*

Using the filter introduces a problem. Because the progress of time is specified in the trajectories, a state of a HTS where only actions are allowed specifies that an action has to be performed without delay. A state of a HTS that allows a trajectories specifies that the actions allowed in that state do not have to take place immediately. Consider the fragments of HTSs in Figure 4.4.

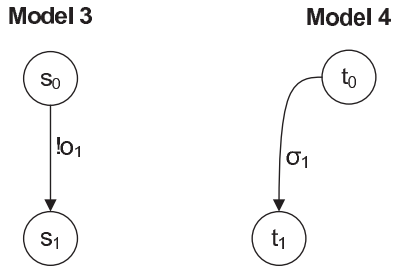

Figure 4.4: Two More Model Fragments

Example 4.10. *In the fragment of* **Model 3** *only o_1 is allowed in state s_0. In the fragment of* **Model 4** *the trajectory σ_1 is allowed in state t_0. Clearly,* **Model 4** *is not supposed to be conform to* **Model 3**, *because if an implementation is not able to perform an output action while according to the specification it is supposed to perform that action, then the implementation should not be considered conform the specification. However, if $V_I \neq \emptyset$ then $\mathbf{infilter}(\{\sigma_1\}, \{\}) = \emptyset$. This makes the fragment of* **Model 4** *conform the fragment of* **Model 3**, *according to the above definition.*

Note that this problem does not occur when besides actions, only continuous output is involved, nor in the timed input-output conformance relation by Krichen and Tripakis [30] described in Chapter 3. The reason is that in those relations we did not need the filter, which could filter out all continuous input behavior allowed by the implementation.

To solve this problem, we introduce a special symbol ξ which indicates that a state is *agile*, meaning that the state does allow time to pass by continuous behavior.

Definition 4.11. *Let $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be an LTS, then a state $s \in S$ is* agile*, denoted by $\xi(s)$, if there exists a $\sigma \in \Sigma$: $s \overset{\sigma}{\rightsquigarrow}$.*

Similar to quiescence in Tretmans's conformance relation we add ξ to the set of allowed output actions.

Definition 4.12. *Let $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a HTS. For a state $s \in S$ we define:*

$$\mathbf{out}(s) = \begin{cases} \{o \in A_O | s \overset{o}{\Rightarrow}\} \cup \{\xi\} & \text{, if } \xi(s); \\ \{o \in A_O | s \overset{o}{\Rightarrow}\} & \text{, otherwise.} \end{cases}$$

For a set of states $C \subseteq S$ we define:

$$\mathbf{out}(C) = \bigcup_{s \in C} \mathbf{out}(s).$$

Finally, the conformance relation for hybrid systems **hioco** is defined using Definitions 4.3, 4.5, 4.8, and 4.12.

Definition 4.13. *Let $\mathcal{S}$ be a HTS and let $\mathcal{I}$ be an input enabled HTS. We say that $\mathcal{I}$ is hybrid input-output conform $\mathcal{S}$, denoted by $\mathcal{I}$ **hioco** $\mathcal{S}$, if and only if for all traces $\alpha \in traces(\mathcal{S})$:*

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \alpha) \wedge$$

$$\mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha), \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha)) \subseteq \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha).$$

According to this definition, **Model 4** is not conform to **Model 3**, for $\mathbf{out}(t_0) \nsubseteq \mathbf{out}(s_0)$, because $\{\xi\} \nsubseteq \{o_1\}$.

4.4 Hybrid Tests

In this section we formalize the notion of test as described in Section 4.1. We define a notion of hybrid test in similar fashion to the notions of test described in Chapter 3.

A hybrid test is a transition system $\mathcal{TC} = (T \cup \{\mathbf{pass}, \mathbf{fail}\}, t_0, A \cup \Sigma_{\mathcal{TC}}, \rightarrow_{\mathcal{TC}} \cup \rightsquigarrow_{\mathcal{TC}})$. It has a tree-like structure (i.e., a test is acyclic), and it has two terminal states **pass** or **fail** as leaves. Hybrid tests are deterministic for actions and deterministic with respect to trajectories. A hybrid test has the following properties.

- The states **pass** and **fail** are terminal states of the test. That is, there does not exist a $\mu \in A \cup \Sigma_{\mathcal{TC}}$ such that $\mathbf{pass} \overset{\mu}{\rightarrow}$ or $\mathbf{fail} \overset{\mu}{\rightarrow}$.

- A test is deterministic with respect to actions. That is, for all $t, t', t'' \in T$ and $a \in A$, if $t \xrightarrow{a}_{\mathcal{TC}} t'$ and $t \xrightarrow{a}_{\mathcal{TC}} t''$, then $t' = t''$.
- A test is deterministic for trajectories in accordance with condition **A2** on HTSs.
- A test maintains trajectory additivity in accordance with condition **A1** on HTSs as well.

Note that a test also allows Zeno behavior. In testing real time systems this behavior is not considered because time always has to progress eventually. This is solved by the test execution algorithm; it has to be implemented such that continuous input is selected such that always time is eventually able to progress.

A test is associated to a specification as follows. If according to the specification some input actions are allowed, the test can allow one of these input actions. If according to the specification some output actions are allowed but no trajectories, which means that the state is not agile, then the allowed output actions may lead to the verdict **pass** or to continuation of the test; the other output actions and all trajectories lead to the verdict **fail**. Therefore if, for a nonempty set of states, the implementation is agile but the specification is not, then the test may lead to the verdict **fail**. If according to the specification trajectories are allowed, a particular input trajectory is chosen. If the complete trajectory (including the value of output variables) is allowed according to the specification, then the test may lead to the verdict **pass** or testing may be continued. All other trajectories lead to the verdict **fail**. It may be that applying the selected input trajectory and observing the output trajectory is interrupted by an output action. If this interruption is allowed according to the specification, then the test may be continued or the verdict **pass** may be given. If the output action was not allowed the verdict **fail** is given.

Definition 4.14. *Let $\mathcal{S} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a specification with continuous variables $V = V_I \uplus V_O$ and let $C \subseteq S$ be a non-empty set of states; then the set of tests, denoted by $hTests(C)$, is inductively defined as follows:*

1. **pass** *is an element of $hTests(C)$.*
2. *Let $i \in A_I$; if C* **after** *$i \neq \emptyset$ and $\mathcal{TC}' \in hTests(C$* **after** *$i)$, then $i;\mathcal{TC}'$ is an element of $hTests(C)$.*
3. *Suppose* $\mathbf{traj}(C) = \emptyset$ *and let, for all $o \in A_O$ with $o \in \mathbf{out}(C)$,* $\mathcal{TC}_o \in hTests(C$ **after** *$o)$; then*

$$\begin{array}{l} \sum\{o;\mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C)\} + \\ \sum\{o;\mathbf{fail} | o \in A_O \backslash \mathbf{out}(C)\} + \\ \sum\{\sigma;\mathbf{fail} | \sigma \in \Sigma\} \end{array}$$

   is an element of $hTests(C)$.

4. *Let $u \in \{\sigma \downarrow V_I | \sigma \in \mathbf{traj}(C)\}$ be a trajectory on input variables.*

- *Denote by* $\mathbf{traj_u}(C) = \{\sigma | \sigma \downarrow V_I = u \wedge \sigma \in \mathbf{traj}(C)\}$ *the set of trajectories with input trajectory* u*, and*
- *denote by* $\mathbf{subtraj_u}(C) = \{\sigma | \exists_{\sigma' \in \mathbf{traj_u}(C)} : \sigma \leq \sigma'\}$ *the set of prefixes of the set of trajectories in* $\mathbf{traj_u}(C)$*.*

Furthermore, let $j = u.ltime$*, let, for all* $\sigma \in \mathbf{traj_u}(C)$*,* $\mathcal{TC}_\sigma \in hTests(C$ **after** $\sigma)$ *and let, for all* $\sigma' \in \mathbf{subtraj_u}(C)$ *and* $o \in \mathbf{out}(C$ **after** $\sigma')$*,* $\mathcal{TC}_{\sigma' o} \in hTests(C$ **after** $\sigma' o)$*. Then*

$$\begin{array}{l}
\sum\{\sigma; TC_\sigma | \sigma \in \mathbf{traj_u}(C)\} + \\
\sum\{\sigma; \mathbf{fail} | \sigma \notin \mathbf{subtraj_u}(C)\} + \\
\sum\{o; \mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C)\} + \\
\sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C)\} + \\
\sum\{\sigma'; o; \mathbf{fail} | \sigma' \in \mathbf{subtraj_u}(C) \wedge \sigma'.ltime < j \wedge o \notin \mathbf{out}(C \textbf{ after } \sigma')\} + \\
\sum\{\sigma'; o; \mathcal{TC}_{\sigma' o} | \sigma' \in \mathbf{subtraj_u}(C) \wedge \sigma'.ltime < j \wedge o \in \mathbf{out}(C \textbf{ after } \sigma')\}
\end{array}$$

is an element of $hTests(C)$*.*

Note that if $V_I = \emptyset$, then u is a trajectory over an empty set of variables, but it is still a trajectory. Therefore, $\mathbf{traj}_u(C)$ contains all trajectories allowed in the set of states C with duration $u.ltime$. Thus, this notion of test also defines tests for hybrid specifications with only continuous output variables, besides actions.

From now on $hTests(\mathcal{S})$ denotes the set of all tests that can be derived from $\mathcal{S}$ starting from the initial state: $hTests(\mathcal{S}) = hTests(\{s_0\})$.

Note that agility ξ is not part of the test. It is tested through case three of the notion of test by selecting any input trajectory and if time is allowed to pass (and the implementation produces output), then the test leads to the verdict **fail**.

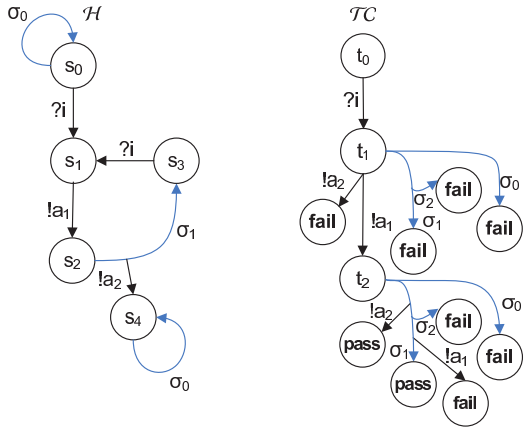

Figure 4.5: Example Hybrid Test

Example 4.15. *Figure 4.5 depicts an illustrative (but incomplete) test $\mathcal{TC}$ generated from a hybrid system $\mathcal{H}$. In our displayed system the transition with output action $!a_2$ means that trajectory σ_1 can be interrupted.*

The example test $\mathcal{TC}$ says to first apply the input action $?i$ to the implementation, and then immediately observe the output action $!a_1$. If an implementation behaves according to σ_1 initially, but behaves according to σ_2 after a while, then the test leads to the verdict **fail** *since σ_2 is not allowed by the specification. At this point σ_1 and σ_0 lead to the verdict fail as well. After $!a_1$, either the complete trajectory σ_1 or the prefix of this trajectory followed by output action $!a_2$ is correct behavior and leads to a verdict* ***pass****. All other behavior leads to the verdict* ***fail****.*

The execution of a hybrid test is defined by the synchronous composition (from Definition 2.7) of the test and the implementation. We do not need to redefine this synchronous composition to include agility, because ξ is not part of implementation or the test.

Definition 4.16. *Let $\mathcal{TC} = (T, t_0, A \cup \Sigma, \rightarrow_{\mathcal{TC}} \cup \rightsquigarrow_{\mathcal{TC}})$ be a test and let $\mathcal{I} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be an implementation. The set of test runs, denoted by $testruns(\mathcal{TC} \parallel \mathcal{I})$, is the set of all traces that lead to a state* **pass** *or* **fail***:*

$$testruns(\mathcal{TC} \parallel \mathcal{I}) =$$

$$\{\alpha \in traces(\mathcal{TC} \parallel \mathcal{I}) | \exists_{s \in S} : (t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{pass}, s) \ \vee \ (t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{fail}, s)\}$$

We say a hybrid implementation passes a hybrid test if only the verdict **pass** is reachable in the run of the test.

Definition 4.17. *Let $\mathcal{TC} = (T, t_0, A \cup \Sigma, \rightarrow_{\mathcal{TC}} \cup \rightsquigarrow_{\mathcal{TC}})$ be a test and let $\mathcal{I} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be an implementation, then $\mathcal{I}$* **passes** *$\mathcal{TC}$ is defined as*

$$\mathcal{I} \textbf{ passes } \mathcal{TC} \iff \forall_{\alpha \in testruns(\mathcal{TC} \parallel \mathcal{I})} : \exists_{s' \in S} : (t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{pass}, s').$$

Let $\mathcal{S}$ be a specification, then

$$\mathcal{I} \textbf{ passes } hTests(\mathcal{S}) \iff \forall_{\mathcal{TC} \in hTests(\mathcal{S})} : \mathcal{I} \textbf{ passes } \mathcal{TC}.$$

4.5 Hybrid Input-output Conformance and Tests with an Environment

For the hybrid conformance relation, the specification does not need to be input enabled. This allows us to restrict the specification and only test the implementation for specific input. The input is part of the specification. However, intuitively, the input for the specification is not part of the specification. E.g. the specification of a thermostat in Example 4.1 defined the behavior of the controller, namely !Error

and !HeaterON messages, plus the behavior of the environment, namely the temperature flow and the ?Reset action. It is more intuitive to specify these behaviors separately, namely in a specification of the controller, which is being tested, and a specification of the environment, which provides the input for the implementation (and the controller specification).

Like Krichen and Tripakis [32], and Larsen et al. [33] have done for their timed conformance relation, it is an option to separate the environment behavior from the specification of the system under test. The idea is to separately specify the environment as a hybrid transition system. The hybrid conformance relation is adapted by replacing the specification in Definition 4.13 by the synchronous composition of an input enabled specification and an environment.

In this case, we want the environment to restrict the input to be applied to the implementation instead of the specification. Therefore, the specification has to be *input complete with respect to* the environment. I.e. after every trace α, all input allowed by the environment after trace α, is also allowed by the specification after trace α. Or, in other words, the input allowed by the environment after trace α is a subset of the input allowed by the specification after trace α.

Definition 4.18. *Let $\mathcal{S} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a specification and let $\mathcal{E} = (E, e_0, A_\tau \cup \Sigma, \rightarrow' \cup \rightsquigarrow')$ be an environment. $\mathcal{S}$ is* input complete with respect to *$\mathcal{E}$ if, for all $\alpha intraces(\mathcal{S} \parallel \mathcal{E})$ holds that*

$$\{i \in A_I | \exists_{e \in (\mathcal{E} \textbf{ after } \alpha)} : e \stackrel{i}{\Rightarrow}\} \subseteq \{i \in A_I | \exists_{s \in (\mathcal{S} \textbf{ after } \alpha)} : s \stackrel{i}{\Rightarrow}\} \wedge$$

$$\{\sigma \downarrow V_I | \sigma \in \Sigma \wedge \exists_{e \in (\mathcal{E} \textbf{ after } \alpha)} : e \stackrel{\sigma}{\Rightarrow}\} \subseteq \{\sigma \downarrow V_I | \sigma \in \Sigma \wedge \exists_{s \in (\mathcal{S} \textbf{ after } \alpha)} : s \stackrel{\sigma}{\Rightarrow}\}.$$

Furthermore, the environment should not prevent the specification (and later on the implementation) from performing output. Therefore, we need to demand that the environment is output enabled. At any moment in time, the environment allows any output action and, for any input trajectory allowed by a state, the environment allows any output trajectory.

Definition 4.19. *A HTS $\mathcal{H}$ is* output enabled *if:*

- *for every $s \in S$ and $o \in A_O$: $s \stackrel{o}{\Rightarrow}$;and*
- *for every $s \in S$:*
  1. *there exists an action $a \in A_I \cup \{\tau\}$ such that $s \stackrel{a}{\rightarrow}$ and there does not exist a $\sigma \in \Sigma$ such that $s \stackrel{\sigma}{\rightsquigarrow}$; or*
  2. *for all $\sigma \in \Sigma$, if $s \stackrel{\sigma}{\rightsquigarrow}$ and $\sigma \downarrow V_I = u$, then for all $v \in trajs(V_O)$ such that $v.ltime = \sigma.ltime$, there exists a $\sigma' \in \Sigma$ such that $s \stackrel{\sigma'}{\rightsquigarrow}$ and $\sigma' \downarrow V_O = v$.*

Note that this is a different notion of enabledness compared to the notion of input enabledness.

Now, if the environment $\mathcal{E}$ is output enabled and the specification $\mathcal{S}$ is input complete with respect to $\mathcal{E}$, then an implementation $\mathcal{I}$ conforms to $\mathcal{S} \parallel \mathcal{E}$ if and only if, for all $\alpha \in traces(\mathcal{S} \parallel \mathcal{E})$,

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha) \wedge$$

$$\mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha), \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha)) \subseteq \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha).$$

The environment specifies which input is considered in the conformance between the specification and the implementation. Moreover, the environment is output enabled. Therefore, we can also define the conformance between a specification and an implementation as the conformance between a specification with environment and an implementation with the same environment. This has the advantage that in the conformance relation the **infilter** is no longer needed because the environment ensures that in both the specification and the implementation the same input is considered.

Definition 4.20. *Let $\mathcal{S}$ and $\mathcal{I}$ be a specification and an implementation respectively. Let $\mathcal{E}$ be an environment. Let $\mathcal{S}$ be input complete with respect to $\mathcal{E}$, let $\mathcal{I}$ be input enabled and let $\mathcal{E}$ be output enabled. $\mathcal{I}$ is input-output conform $\mathcal{S}$ with respect to environment $\mathcal{E}$, denoted by $\mathcal{I}$ $\mathbf{hioco}_{\mathcal{E}}$ $\mathcal{S}$, if and only if for all $\alpha \in traces(\mathcal{S} \parallel \mathcal{E})$:*

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha) \wedge$$

$$\mathbf{traj}(\mathcal{I} \parallel \mathcal{E} \textbf{ after } \alpha) \subseteq \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha).$$

Theorem 4.21. *: Let $\mathcal{S}$ be a specification and let $\mathcal{I}$ be an implementation and let $\mathcal{E}$ be an environment. Let $\mathcal{E}$ be not blocking for $\mathcal{S}$ and $\mathcal{I}$; then $\mathcal{I}$ $\mathbf{hioco}$ $\mathcal{S} \parallel \mathcal{E}$ if and only if $\mathcal{I}$ $\mathbf{hioco}_{\mathcal{E}}$ $\mathcal{S}$.*

Sketch of proof. We prove this theorem by showing that $\mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha), \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha))$ and $\mathbf{traj}(\mathcal{I} \parallel \mathcal{E} \textbf{ after } \alpha)$ are the same sets of trajectories.

- The set of trajectories $\mathbf{traj}(\mathcal{I} \parallel \mathcal{E} \textbf{ after } \alpha)$ is the set $\{\sigma \in \Sigma_{\mathcal{I}\parallel\mathcal{E}} | \mathcal{I} \parallel \mathcal{E} \stackrel{\alpha\sigma}{\Rightarrow}\}$. By the definition of synchronous composition this is the set:

  $$\{\sigma \in \Sigma_{\mathcal{I}} | \mathcal{I} \stackrel{\alpha\sigma}{\Rightarrow} \wedge \mathcal{E} \stackrel{\alpha\sigma}{\Rightarrow}\}$$

- The set of trajectories $\mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha), \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha))$ is the set

  $$\mathbf{infilter}(\{\sigma \in \Sigma_{\mathcal{I}} | \mathcal{I} \stackrel{\alpha\sigma}{\Rightarrow}\}, \{\sigma \in \Sigma_{\mathcal{S}\parallel\mathcal{E}} | \mathcal{S} \parallel \mathcal{E} \stackrel{\alpha\sigma}{\Rightarrow}\}).$$

  By the definition of **infilter** this is the set

  $$\{\sigma \in \Sigma_{\mathcal{I}} | \mathcal{I} \stackrel{\alpha\sigma}{\Rightarrow} \wedge \exists_{\sigma' \in \Sigma_{\mathcal{S}\parallel\mathcal{E}}} : \sigma \downarrow V_I = \sigma' \downarrow V_I \wedge \mathcal{S} \parallel \mathcal{E} \stackrel{\alpha\sigma'}{\Rightarrow}\},$$

which can be rewritten as

$$\{\sigma \in \Sigma_{\mathcal{I}} | \mathcal{I} \overset{\alpha\sigma}{\Rightarrow} \wedge \exists_{\sigma' \in \Sigma_{\mathcal{S} \parallel \mathcal{E}}} : \sigma \downarrow V_I = \sigma' \downarrow V_I \wedge \mathcal{S} \overset{\alpha\sigma'}{\Rightarrow} \wedge \mathcal{E} \overset{\alpha\sigma'}{\Rightarrow}\}.$$

Because $\mathcal{S}$ is input complete with respect to $\mathcal{E}$ it holds that if $\sigma' \downarrow V_I$ is an input trajectory of $\mathcal{S}$ after trace α then $\sigma' \downarrow V_I$ is an input trajectory of a $\mathcal{E}$ after trace α. Therefore:

$$\{\sigma \in \Sigma_{\mathcal{I}} | \mathcal{I} \overset{\alpha\sigma}{\Rightarrow} \wedge \exists_{\sigma' \in \Sigma_{\mathcal{E}}} : \sigma \downarrow V_I = \sigma' \downarrow V_I \wedge \mathcal{E} \overset{\alpha\sigma'}{\Rightarrow}\}.$$

Finally, because $\mathcal{E}$ is output enabled, it contains all output trajectories in $\Sigma_{\mathcal{I}}$ and thus:

$$\{\sigma \in \Sigma_{\mathcal{I}} | \mathcal{I} \overset{\alpha\sigma}{\Rightarrow} \wedge \mathcal{E} \overset{\alpha\sigma}{\Rightarrow}\}.$$

Then, it follows that, for all $\alpha \in traces(\mathcal{S} \parallel \mathcal{E})$:

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha) \wedge$$

$$\mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha), \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha)) \subseteq \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha)$$

if and only if

$$\mathbf{out}(\mathcal{I} \textbf{ after } \alpha) \subseteq \mathbf{out}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha) \wedge$$

$$\mathbf{traj}(\mathcal{I} \parallel \mathcal{E} \textbf{ after } \alpha) \subseteq \mathbf{traj}(\mathcal{S} \parallel \mathcal{E} \textbf{ after } \alpha).$$

Using a separate environment does not change the definition of a hybrid test. The set of tests that can be generated from a specification $\mathcal{S}$ with environment $\mathcal{E}$ is defined by $hTests(\mathcal{S} \parallel \mathcal{E})$.

4.6 Soundness and Exhaustiveness Proofs

In this section we prove the soundness and exhaustiveness of our hybrid tests. Throughout this section the HTS $\mathcal{S}$ is defined by the tuple $(S_{\mathcal{S}}, s_0, A_\tau \cup \Sigma, \rightarrow_{\mathcal{S}} \cup \rightsquigarrow_{\mathcal{S}})$ and the HTS $\mathcal{I}$ is defined by the tuple $(S_{\mathcal{I}}, i_0, A_\tau \cup \Sigma, \rightarrow_{\mathcal{I}} \cup \rightsquigarrow_{\mathcal{I}})$.

Even though the complete proof comprises over six pages, the proof ideas are the same as for the soundness and exhaustive proofs by Tretmans (see Chapter 3). Soundness is proven by contraposition: we suppose that there is a test that leads to the verdict fail for an implementation, and we prove that such a test can only exist if the implementation does not conform to the specification. First we prove that for every test run that leads to the verdict **fail**, there is a normalized trace and a subtest such that by executing one trajectory, one output action, or one trajectory followed by an output action, the verdict fail can be reached. Using the structure of the subtest we then prove that this implies that if a test run leads to **fail**, then this is because of an output action or a trajectory that is not allowed by the specification. However, since the test run leads to **fail**, this means that

this output action or trajectory was allowed by the implementation. Therefore, the implementation does not conform to the specification, which contradicts the assumption that the implementation does conform to the specification.

Exhaustiveness is proven by constructing a test from a trace that has to lead to the verdict **fail** for an implementation that does not conform to the specification.

First we define the notion of subtest. A subtest is (the rest of) a test after a trace in the shape of a test.

Definition 4.22. *Let S be a specification, and let S be the set of states of $\mathcal{S}$. Let $C \subseteq S$ be a set of states. Let $\mathcal{TC} = (T, t_0, A \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a test, with $\mathcal{TC} \in hTests(C)$. Let $t \in T$ be state of $\mathcal{TC}$. Let $\alpha \in traces(\mathcal{TC})$ and let $t_0 \stackrel{\alpha}{\Rightarrow} t$. Then, the* subtest $\mathcal{TC}' \in hTests(C$ **after** $\alpha)$ *is the test $\mathcal{TC}' = (T', t, A \cup \Sigma, \rightarrow' \cup \rightsquigarrow')$, with*

- $T' = \{t' | t' \in T \wedge \exists_{\beta \in (A \cup \Sigma)^*} : t \stackrel{\beta}{\Rightarrow} t'\}$
- $\rightarrow' = \{(t', a, t'') | t', t'' \in T \wedge a \in A \wedge \exists_{\beta \in (A \cup \Sigma)^*} : t \stackrel{\beta}{\Rightarrow} t' \wedge t' \stackrel{a}{\rightarrow} t''\}$*; and*
- $\rightsquigarrow' = \{(t', \sigma, t'') | t', t'' \in T \wedge \sigma \in \Sigma \wedge \exists_{\beta \in (A \cup \Sigma)^*} : t \stackrel{\beta}{\Rightarrow} t' \wedge t' \stackrel{\sigma}{\rightarrow} t''\}$.

Let $\mathcal{TC}$ be a test. Let $t_0 \in T$ be the initial state of $\mathcal{TC}$. Then we write $\mathcal{TC} \stackrel{a}{\rightarrow}$ **fail** if $t_0 \stackrel{a}{\rightarrow}$ **fail**, and $\mathcal{TC} \stackrel{\sigma}{\rightsquigarrow}$ **fail** if $t_0 \stackrel{\sigma}{\rightsquigarrow}$ **fail**, and $\mathcal{TC} \stackrel{\alpha}{\Rightarrow}$ **fail** if $t_0 \stackrel{\alpha}{\Rightarrow}$ **fail**. Let $\mathcal{TC}'$ be a subtest of $\mathcal{TC}$ and let $t_0' \in T$ be the initial state of $\mathcal{TC}'$. Then we write $\mathcal{TC} \stackrel{a}{\rightarrow} \mathcal{TC}'$ if $t_0 \stackrel{a}{\rightarrow} t_0'$ in $\mathcal{TC}$, $\mathcal{TC} \stackrel{\sigma}{\rightsquigarrow} \mathcal{TC}'$ if $t_0 \stackrel{\sigma}{\rightsquigarrow} t_0'$ in $\mathcal{TC}$, and $\mathcal{TC} \stackrel{\alpha}{\Rightarrow} \mathcal{TC}'$ if $t_0 \stackrel{\alpha}{\Rightarrow} t_0'$. From now on, if a trace α does not fit on a generalized transition, then we also write $s \stackrel{\alpha}{\Longrightarrow} s'$ if $s \stackrel{\alpha}{\Rightarrow} s'$.

A normalized trace is a trace with all sequences of consecutive trajectories concatenated into one trajectory.

Definition 4.23. *Let $\alpha \in (A \cup \Sigma)^*$ be sequence of labels. Then a* normalized trace*, denoted by* $\mathbf{ntrace}(\alpha)$*, is defined as:*

- *if $\alpha = \epsilon$ then* $\mathbf{ntrace}(\alpha) = \epsilon$
- *if $\alpha = a$, for some action a, then* $\mathbf{ntrace}(\alpha) = a$*;*
- *if $\alpha = \sigma$ then* $\mathbf{ntrace}(\alpha) = \sigma$*;*
- *if $\alpha = a\alpha'$ then* $\mathbf{ntrace}(\alpha) = a\ \mathbf{ntrace}(\alpha')$*;*
- *if $\alpha = \sigma\alpha'$ and* $\mathbf{ntrace}(\alpha') = a\alpha''$ *then* $\mathbf{ntrace}(\alpha) = \sigma\ a\ \mathbf{ntrace}(\alpha')$*; and*
- *if $\alpha = \sigma\alpha'$ and* $\mathbf{ntrace}(\alpha') = \sigma\alpha''$ *then* $\mathbf{ntrace}(\alpha) = \sigma \frown \sigma'\ \mathbf{ntrace}(\alpha'')$.

Because we posed trajectory additivity on HTSs, we know that for every trace α in an HTS there exists a normalized trace as well, leading to the same state.

Lemma 4.24. *Let α be a trace and let $\mathcal{H}$ be a hybrid transition system. If $s_0 \stackrel{\alpha}{\Rightarrow} s$, then $s_0 \stackrel{\mathbf{ntrace}(\alpha)}{\Longrightarrow} s$.*

This lemma holds because of trajectory additivity, which we posed on HTSs (see condition **A1** in Section 2.1.3.

Lemma 4.25. *If $\mathcal{TC} \in hTests(C)$ and $\alpha \in testruns(\mathcal{TC} \parallel \mathcal{I})$ such that $\mathcal{TC} \stackrel{\alpha}{\Rightarrow}$ **fail**, then there exists a trace $\alpha' \in traces(C)$, a subtest $\mathcal{TC}'$, and an output action $o \in A_O$ and a trajectory σ such that $\mathcal{TC}' \in hTests(C$ **after** $\alpha')$, $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$, and either:*

1. $\mathbf{ntrace}(\alpha) = \alpha' o$ *and* $\mathcal{TC}' \stackrel{o}{\rightarrow}$ **fail**;
2. $\mathbf{ntrace}(\alpha) = \alpha' \sigma$ *and* $\mathcal{TC}' \stackrel{\sigma}{\rightsquigarrow}$ **fail**; *or*
3. $\mathbf{ntrace}(\alpha) = \alpha' \sigma o$ *and* $\mathcal{TC}' \stackrel{\sigma o}{\Rightarrow}$ **fail**.

Proof. We prove this lemma by induction on the structure of $\mathcal{TC}$.

1. Suppose that $\mathcal{TC} =$ **pass**. Then $\mathcal{TC} \stackrel{\alpha}{\Rightarrow}$ **fail** is not possible, so the lemma vacuously holds.
2. Suppose that $\mathcal{TC} = i; \mathcal{TC}'$, with $i \in A_I$ and $\mathcal{TC}' \in hTests(C$ **after** $i)$, then $\alpha = i\alpha'$. By the inducation hypothesis Lemma 4.25 holds for $\mathcal{TC}' and \alpha'$. So, $\mathcal{TC} \stackrel{i}{\rightarrow} \mathcal{TC}'$ and since, by the induction hypothesis, Lemma 4.25 holds for $\mathcal{TC}'$ and α', it follows that Lemma 4.25 holds for $\mathcal{TC}$ and α.
3. Suppose that
$$\begin{aligned}\mathcal{TC} = &\textstyle\sum\{o; \mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C)\} + \\ &\textstyle\sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C)\} + \\ &\textstyle\sum\{\sigma; \mathbf{fail} | \sigma \in \Sigma\},\end{aligned}$$
with $o \in A_O$ and, for all $o \in \mathbf{out}(C)$, $\mathcal{TC}_o \in hTests(C$ **after** $o)$. Suppose that Lemma 4.25 holds for all $\mathcal{TC}_o$ and every $\alpha' \in testruns(\mathcal{TC}_o \parallel \mathcal{I})$ for which $\mathcal{TC}' \stackrel{\alpha'}{\Rightarrow}$ **fail** (the induction hypothesis). $\mathcal{TC}_o$ is a subtest of $\mathcal{TC}$. From the shape of $\mathcal{TC}$ it follows that we can distinguish three cases:
   (a) suppose $\alpha = o\,\alpha'$, with $o \in A_O \cap \mathbf{out}(C)$, and $\alpha' \in testruns(\mathcal{TC}_o \parallel \mathcal{I})$; then $\mathcal{TC} \stackrel{o}{\rightarrow} \mathcal{TC}_o$ and since, by the induction hypothesis, Lemma 4.25 holds for $\mathcal{TC}_o$ and α', it follows that Lemma 4.25 holds for $\mathcal{TC}$ and α;
   (b) suppose $\alpha = o$, with $o \in A_O \backslash \mathbf{out}(C)$, then, $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{o}{\rightarrow}$ **fail**, and it follows that case 1 of Lemma 4.25 holds for $\mathcal{TC}$ and α; or
   (c) suppose $\alpha = \sigma$, then, $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{\sigma}{\rightsquigarrow}$ **fail**, and it follows that case 2 of Lemma 4.25 holds for $\mathcal{TC}$ and α.

4. Suppose that

$$\begin{aligned}\mathcal{TC} = \sum\{\sigma; \mathcal{TC}_\sigma | \sigma \in \mathbf{traj_u}(C)\} \ + \sum\{\sigma; \mathbf{fail} | \sigma \notin \mathbf{subtraj_u}(C)\} + \\ \sum\{o; \mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C)\} + \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C)\} + \\ \sum\{\sigma'; o; \mathcal{TC}_{\sigma' o} | \sigma' \in \mathbf{subtraj_u}(C) \wedge \sigma'.ltime < j \wedge \\ o \in \mathbf{out}(C \textbf{ after } \sigma')\} + \\ \sum\{\sigma'; o; \mathbf{fail} | \sigma' \in \mathbf{subtraj_u}(C) \wedge \sigma'.ltime < j \wedge \\ o \notin \mathbf{out}(C \textbf{ after } \sigma')\}.\end{aligned}$$

. Here $\mathcal{TC}_o$, $\mathcal{TC}_\sigma$, and $\mathcal{TC}_{\sigma' o}$ are subtests of $\mathcal{TC}$. Suppose that Lemma 4.25 holds for every $\mathcal{TC}_\sigma$, $\mathcal{TC}_o$, and $\mathcal{TC}_{\sigma o}$ and every $\alpha' \in testruns(\mathcal{TC}_\sigma \parallel \mathcal{I}) \cup testruns(\mathcal{TC}_o \parallel \mathcal{I}) \cup testruns(\mathcal{TC}_{\sigma o} \parallel \mathcal{I})$ (the induction hypothesis). From the shape of $\mathcal{TC}$ it follows that we can distinguish six cases:

(a) if $\alpha = \sigma\ \alpha'$, with $\sigma \in \mathbf{subtraj_u}(C)$ and $\alpha' \in testruns(\mathcal{TC}_\sigma \parallel \mathcal{I})$, then:

i. suppose $\mathbf{ntrace}(\alpha') = \sigma''$ and $\sigma \frown \sigma'' \notin \mathbf{traj_u}(C)$ then $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{\sigma \frown \sigma''}{\rightsquigarrow}$ **fail**, and it follows that case 2 of Lemma 4.25 holds for $\mathcal{TC}$ and α;

ii. if $\mathbf{ntrace}(\alpha') = \sigma'' o$, then $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{\sigma \frown \sigma'' o}{\Longrightarrow}$ **fail**, and it follows that case 3 of Lemma 4.25 holds for $\mathcal{TC}$ and α; and

iii. in any other case (i.e. $\mathbf{ntrace}(\alpha') = o\alpha''$, or $\mathbf{ntrace}(\alpha') = \sigma''\alpha''$ and $\alpha'' \neq \epsilon$) then $\mathcal{TC} \stackrel{\sigma}{\rightsquigarrow} \mathcal{TC}_\sigma$ and, by the induction hypothesis, since Lemma 4.25 holds for $\mathcal{TC}_\sigma$ and α', it follows that Lemma 4.25 holds for $\mathcal{TC}$ and α;

(b) suppose $\alpha = \sigma$, with $\sigma \notin \mathbf{subtraj_u}(C)$, then $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{\sigma}{\rightsquigarrow}$ **fail**, and it follows that case 2 of Lemma 4.25 holds for $\mathcal{TC}$ and α;

(c) suppose $\alpha = o\ \alpha'$, with $o \in A_O \cap \mathbf{out}(C)$ and $\alpha' \in testruns(\mathcal{TC}_o \parallel \mathcal{I})$, then $\mathcal{TC} \stackrel{o}{\rightarrow} \mathcal{TC}_o$ and since, by the induction hypothesis), Lemma 4.25 holds for $\mathcal{TC}_o$ and α', it follows that Lemma 4.25 holds for $\mathcal{TC}$ and α;

(d) suppose $\alpha = o$, with $o \in A_O \backslash \mathbf{out}(C)$, then $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{o}{\rightarrow}$ **fail**, and it follows that case 1 of Lemma 4.25 holds for $\mathcal{TC}$ and α;

(e) suppose $\alpha = \sigma' o\ \alpha$) with $\sigma' \in \mathbf{subtraj_u}(C)$, $o \in \mathbf{out}(C \textbf{ after } \sigma')$ and $\alpha' \in testruns(\mathcal{TC}_{\sigma' o} \parallel \mathcal{I})$, then $\mathcal{TC} \stackrel{\sigma' o}{\Rightarrow} \mathcal{TC}_{\sigma' o}$ and, by the induction hypothesis, since Lemma 4.25 holds for $\mathcal{TC}_{\sigma' o}$ and α', it follows that Lemma 4.25 holds for $\mathcal{TC}$ and α; and

(f) suppose $\alpha = \sigma' o$, with $\sigma' \in \mathbf{subtraj_u}(C)$ and $o \notin \mathbf{out}(C \textbf{ after } \sigma')$, then $\mathcal{TC} \stackrel{\epsilon}{\Rightarrow} \mathcal{TC}$ and $\mathcal{TC} \stackrel{\sigma' o}{\Rightarrow} \mathcal{TC}_{\sigma' o}$, and it follows that case 2 of Lemma 4.25 holds for $\mathcal{TC}$ and α.

If a normalized trace leads to the verdict **fail**, then that is either because it ends with an output that was not allowed by the specification or it ends with a trajectory that was not allowed by the specification.

Lemma 4.26. *If $\mathcal{TC} \in hTests(\mathcal{S})$ and $\alpha \in testruns(\mathcal{TC} \parallel \mathcal{I})$ such that $\mathcal{TC} \stackrel{\alpha}{\Rightarrow}$ **fail**, then:*

- *if* $\mathbf{ntrace}(\alpha) = \alpha' o$, *then* $o \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$; *and*
- *if* $\mathbf{ntrace}(\alpha) = \alpha'\sigma$, *then* $\sigma \notin \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha')$.

Proof. We separately prove the two cases.

- Suppose that $\mathbf{ntrace}(\alpha) = \alpha' o$. According to Lemma 4.25 there are two cases to distinguish.
  1. Suppose $\mathbf{ntrace}(\alpha) = \alpha' o$, and there exists a subtest $\mathcal{TC}' \in hTests(\mathcal{S}\mathbf{after}\ \alpha')$ such that $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$ and $\mathcal{TC}' \stackrel{o}{\rightarrow} \mathbf{fail}$. By definition of $hTests(\mathcal{S}\ \mathbf{after}\ \alpha')$, we know that $\mathcal{TC}'$ is described by either case 3 or case 4 of our notion of test . By this notion $\mathcal{TC}' \stackrel{o}{\rightarrow} \mathbf{fail}$, only if $o \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$.
  2. Suppose $\mathbf{ntrace}(\alpha) = \alpha''\sigma o$, and there exists a subtest $\mathcal{TC}'' \in hTests(\mathcal{S}\ \mathbf{after}\ \alpha'')$ such that $\mathcal{TC} \stackrel{\alpha''}{\Rightarrow} \mathcal{TC}''$ and $\mathcal{TC}'' \stackrel{\sigma o}{\Rightarrow} \mathbf{fail}$. Suppose $\mathcal{TC}''$ is the last subtest of $\mathcal{TC}$ for trace α. That is, there does not exist a subtest $\mathcal{TC}'''$ such that, with $\sigma = \sigma_1 \frown \sigma_2$, and $\mathcal{TC}'' \stackrel{\sigma_1}{\Rightarrow} \mathcal{TC}'''$ and $\mathcal{TC}''' \stackrel{\sigma_2 o}{\Rightarrow} \mathbf{fail}$. We know that there exists such $\mathcal{TC}''$ because of Lemma 2. In this case we know that $\mathcal{TC}''$ is described by case 4 of our notion of test and $\sigma \in \mathbf{subtraj_u}(S\ \mathbf{after}\ \alpha'')$ and $o \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha''\sigma)$. Because $\alpha' = \mathbf{ntrace}(\alpha''\sigma)$ we conclude that $o \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$.
- Suppose that $\mathbf{ntrace}(\alpha) = \alpha'\sigma$. Suppose that $\mathcal{TC}'' \in hTests(\mathcal{S}\ \mathbf{after}\ \alpha'')$ is the last subtest of $\mathcal{TC}$ for trace α. That is, supposing $\mathbf{ntrace}(\alpha) = \mathbf{ntrace}(\alpha''\sigma'')$, $\mathcal{TC} \stackrel{\alpha''}{\Rightarrow} \mathcal{TC}''$ and $\mathcal{TC}'' \stackrel{\sigma''}{\rightsquigarrow} \mathbf{fail}$ and there does not exist a subtest $\mathcal{TC}'''$ such that $\sigma'' = \sigma_1'' \frown \sigma_2''$ and $\mathcal{TC}'' \stackrel{\sigma_1''}{\Rightarrow} \mathcal{TC}'''$ and $\mathcal{TC}''' \stackrel{\sigma_2''}{\rightsquigarrow} \mathbf{fail}$. Again, we know that there exists such $\mathcal{TC}''$ (since, according to Lemma 4.25, $\mathcal{TC}'$ could be this last subtest). In this case $\mathcal{TC}''$ is described by case 3 or case 4 of our test definition and either $\sigma'' \notin \Sigma$ or $\sigma'' \notin \mathbf{subtraj_u}(\mathcal{S}\ \mathbf{after}\ \alpha'')$, which means $\sigma'' \notin \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha'')$. By condition **A2** on our hybrid transition systems we conclude that in this case also $\sigma \notin \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha')$.

The following theorem states that if an implementation conforms to a specification, all test runs of all tests (associated to the specification by our inductive definition) lead to verdict **pass**. In line with other conformance theories we call this soundness of our tests.

Theorem 4.27. *If* $\mathcal{S}$ *is a specification and* $\mathcal{I}$ *is an implementation, then:*

$$\mathcal{I}\ \mathbf{hioco}\ \mathcal{S} \implies \forall_{\mathcal{TC} \in hTests(\{s_0\})} : \mathcal{I}\ \mathbf{passes}\ \mathcal{TC}.$$

Proof. We prove the theorem by contraposition. Suppose that $\mathcal{TC} \in hTests(\{s_0\})$ and $\alpha \in testruns(\mathcal{TC} \parallel \mathcal{I})$ and $s' \in S_{\mathcal{I}}$ and $(t_0, s_0) \stackrel{\alpha}{\Rightarrow} (\mathbf{fail}, s')$. We show that this α can only exist if $\mathcal{I}$ $\mathbf{hi\phi co}$ $\mathcal{S}$.

From Lemma 4.24 we know that if $\mathcal{TC} \stackrel{\alpha}{\Rightarrow} \mathbf{fail}$ then $\mathcal{TC} \stackrel{\mathbf{ntrace}(\alpha)}{\Longrightarrow} \mathbf{fail}$.

- Suppose that $\mathbf{ntrace}(\alpha) = \alpha' o$. Then from Lemma 4.26 we know that $o \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha')$. Because $\mathbf{ntrace}(\alpha) \in testruns(\mathcal{TC} \parallel \mathcal{I})$ we also know that $o \in (\mathcal{I}\ \mathbf{after}\ \alpha')$ (by definition of synchronous composition). Therefore,

$$\mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha') \not\subseteq \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha').$$

- Suppose that $\mathbf{ntrace}(\alpha) = \alpha'\sigma$. Then from Lemma 4.26 we know that $\sigma \notin \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha')$. Because $\mathbf{ntrace}(\alpha) \in testruns(\mathcal{TC} \parallel \mathcal{I})$ we also know that $\sigma \in (\mathcal{I}\ \mathbf{after}\ \alpha')$ (by definition of synchronous composition). From Lemma 4.25 we know there exists a subtest $\mathcal{TC}' \in hTests(\mathcal{S}\ \mathbf{after}\ \alpha')$ such that $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$ and $\mathcal{TC} \stackrel{\sigma}{\rightsquigarrow} \mathbf{fail}$. By the structure of α, we know that $\mathcal{TC}'$ is either described by case 3 or case 4 of the hybrid notion of test (see Definition 4.14).
  - Suppose that $\mathcal{TC}'$ is a test described by case 3 of Definition 4.14. Then $\mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha') = \emptyset$ which means $\xi \notin \mathbf{out}(S\ \mathbf{after}\ \alpha')$. However, since $\mathbf{traj}(\mathcal{I}\ \mathbf{after}\ \alpha') \neq \emptyset$, $\xi \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha')$. Therefore,

$$\mathbf{out}(\mathcal{I}\ \mathbf{after}\ \alpha') \not\subseteq \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \alpha').$$

  - Suppose that $\mathcal{TC}'$ is a test described by case 4 of the Definition 4.14. Because, by the definition of tests, the input behavior is always selected from the specification:

$$\sigma \in \mathbf{infilter}(\mathbf{traj}(\mathcal{I}\ \mathbf{after}\ \alpha'), \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha')).$$

    Therefore,

$$\mathbf{infilter}(\mathbf{traj}(\mathcal{I}\ \mathbf{after}\ \alpha), \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha')) \not\subseteq \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha').$$

We conclude that, if $\alpha \in testruns(\mathcal{TC} \parallel \mathcal{I})$ and $\mathcal{TC} \stackrel{\alpha}{\Rightarrow} \mathbf{fail}$, then $\mathcal{I}\ \mathbf{hi\not{o}co}\ \mathcal{S}$. This contradicts the assumption that $\mathcal{I}\ \mathbf{hioco}\ \mathcal{S}$.

The following theorem states that if an implementation is not conform a specification ($\mathcal{I}\ \mathbf{hi\not{o}co}\ \mathcal{S}$), then there is a test $\mathcal{TC}$ which leads to verdict **fail** ($\mathcal{I}\ \mathbf{pa\not{s}ses}\ \mathcal{TC}$). In line with the **ioco** theory we call this exhaustiveness of our tests.

Theorem 4.28. *If $\mathcal{S}$ is a specification and $\mathcal{I}$ is an implementation, then:*

$$\mathcal{I}\ \mathbf{hi\not{o}co}\ \mathcal{S} \implies \exists_{\mathcal{TC} \in hTests(\mathcal{S})} : \mathcal{I}\ \mathbf{pa\not{s}ses}\ \mathcal{TC}.$$

Proof. Let $C_{\mathcal{I}} \subseteq S_{\mathcal{I}}$ and $C_{\mathcal{S}} \subseteq S_{\mathcal{S}}$. We prove that if

$$\mathbf{out}(C_{\mathcal{I}}\ \mathbf{after}\ \alpha) \not\subseteq \mathbf{out}(C_{\mathcal{S}}\ \mathbf{after}\ \alpha) \vee$$

$$\mathbf{infilter}(\mathbf{traj}(C_{\mathcal{I}}\ \mathbf{after}\ \alpha), \mathbf{traj}(C_{\mathcal{S}}\ \mathbf{after}\ \alpha)) \not\subseteq \mathbf{traj}(C_{\mathcal{S}}\ \mathbf{after}\ \alpha).$$

then there exists a $\mathcal{TC} \in hTests(C_{\mathcal{S}})$ with initial state t_0 and $\alpha\mu \in testruns(\mathcal{TC} \parallel \mathcal{I})$ and, with $c \in C_{\mathcal{I}}$ and $c' \in C\ \mathbf{after}\ \alpha\mu$, $(t_0, c) \stackrel{\alpha\mu}{\Rightarrow} (\mathbf{fail}, c')$. Then, the theorem follows for $C_{\mathcal{I}} = \{s_{0\mathcal{I}}\}$ and $C_{\mathcal{S}} = \{s_{0\mathcal{S}}\}$. We proceed by induction on the length of α.

- Suppose that $\alpha = \epsilon$. Then,

$$\mathbf{out}(C_{\mathcal{I}}) \not\subseteq \mathbf{out}(C_{\mathcal{S}}) \vee \mathbf{infilter}(\mathbf{traj}(C_{\mathcal{I}}), \mathbf{traj}(C_{\mathcal{S}})) \not\subseteq \mathbf{traj}(C_{\mathcal{S}}).$$

  - Suppose that there exists an $o \in \mathbf{out}(C_{\mathcal{I}}) \backslash \mathbf{out}(C_{\mathcal{S}}) \cap A_O$.
    - Suppose that $\mathbf{traj}(C_{\mathcal{S}}) = \emptyset$ Then, for instance, $\mathcal{TC} =$

$$\begin{array}{l} \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma; \mathbf{fail} | \sigma \in \Sigma\} \end{array}$$

      leads to verdict **fail** (since $o \notin \mathbf{out}(C_{\mathcal{S}})$), and therefore $\mathcal{TC} \xrightarrow{o} \mathbf{fail}$.
    - Suppose that $\mathbf{traj}(C_{\mathcal{S}}) \neq \emptyset$. Then, for instance, with some $u \in \{\sigma \downarrow V_I | \sigma \in \mathbf{traj}(C_{\mathcal{S}})\}$ and $j = u.ltime$, $\mathcal{TC} =$

$$\begin{array}{l} \sum\{\sigma; \mathbf{pass} | \sigma \in \mathbf{traj_u}(C_{\mathcal{S}})\} + \sum\{\sigma; \mathbf{fail} | \sigma \notin \mathbf{subtraj_u}(C_{\mathcal{S}})\} + \\ \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma'; o; \mathbf{pass} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \in \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\} + \\ \sum\{\sigma'; o; \mathbf{fail} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \notin \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\} \end{array}$$

      leads to verdict **fail** (since $o \notin \mathbf{out}(C_{\mathcal{S}})$), and therefore $\mathcal{TC} \xrightarrow{o} \mathbf{fail}$.

    In both cases, since $o \in \mathbf{out}(C_{\mathcal{I}})$, $(t'_0, c) \stackrel{o}{\Rightarrow} (\mathbf{fail}, c')$.
  - Suppose that there exists a $\sigma \in \mathbf{infilter}(\mathbf{traj}(C_{\mathcal{I}}), \mathbf{traj}(C_{\mathcal{S}})) \backslash \mathbf{traj}(C_{\mathcal{S}})$. Then, for instance, with $u = \sigma \downarrow V_I$ and $j = u.ltime$, $\mathcal{TC} =$

$$\begin{array}{l} \sum\{\sigma; \mathbf{pass} | \sigma \in \mathbf{traj_u}(C_{\mathcal{S}})\} + \sum\{\sigma; \mathbf{fail} | \sigma \notin \mathbf{subtraj_u}(C_{\mathcal{S}})\} + \\ \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma'; o; \mathbf{pass} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \in \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\} + \\ \sum\{\sigma'; o; \mathbf{fail} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \notin \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\} \end{array}$$

    leads to verdict **fail** (since $\sigma \notin \mathbf{traj_u}(C_{\mathcal{S}})$). Hence, since $\sigma \in \mathbf{traj}(C_{\mathcal{I}})$, $(t'_0, c) \stackrel{\sigma}{\Rightarrow} (\mathbf{fail}, c')$.
  - Suppose that $\xi \in \mathbf{out}(C_{\mathcal{I}}) \backslash \mathbf{out}(C_{\mathcal{S}})$. In this case we know that

$$\mathbf{traj}(C_{\mathcal{S}}) = \emptyset \text{ and } \mathbf{traj}(C_{\mathcal{I}}) \neq \emptyset.$$

    Then, for instance, $\mathcal{TC} =$

$$\begin{array}{l} \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma; \mathbf{fail} | \sigma \in \Sigma\} \end{array}$$

    leads to verdict **fail** for any $\sigma \in \Sigma$. Hence, for any $\sigma \in \mathbf{traj}(C_{\mathcal{I}})$, $(t'_0, c) \stackrel{\sigma}{\Rightarrow} (\mathbf{fail}, c')$.

- Suppose that $\alpha = \nu\alpha'$, with $\alpha' \neq \epsilon$ and (the induction hypothesis): there exists a $\mathcal{TC} \in hTests(C_{\mathcal{S}}$ **after** $\nu)$ with initial state t_0 and $\alpha\mu \in testruns(\mathcal{TC} \parallel \mathcal{I})$ and, with $c \in C_{\mathcal{I}}$ and $c' \in C_{\mathcal{I}}$ **after** $\alpha\mu$, $(t_0, c) \stackrel{\alpha\mu}{\Rightarrow} (\mathbf{fail}, c')$.

  We construct a test $\mathcal{TC}'$ with $\nu\alpha' \in testruns(\mathcal{TC}' \parallel \mathcal{I})$ and $\alpha' \neq \epsilon$. Note that by definition of our notion of test we know that either $\nu \in A_I$, $\nu \in \mathbf{out}(C_{\mathcal{S}}) \cap A_O$, or $\nu \in \mathbf{traj}(C_{\mathcal{S}})$. Otherwise, $\nu\alpha'$ would not be a testrun because after the verdict **fail**, the test terminates.

  - Suppose that $\nu = i$, with $i \in A_I$. Then, the test $i; \mathcal{TC}$ will also lead to verdict **fail** for trace α.
  - Suppose $\nu = o$, with $o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})$, and $\mathbf{traj}(\mathcal{S}) = \emptyset$. Then, for instance, the test

    $$\begin{array}{l}\sum\{o'; \mathcal{TC}_{o'} | o' \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{o'; \mathbf{fail} | o' \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma; \mathbf{fail} | \sigma \in \Sigma\}\end{array}$$

    with $\mathcal{TC}_o = \mathcal{TC}$ and, for $o' \in (A_O \cap \mathbf{out}(C_{\mathcal{S}}))\backslash\{o\}$, $\mathcal{TC}_{o'} =$ **pass** will also lead to verdict **fail** for trace α.
  - Suppose that $\nu = o$, with $o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})$, and $\mathbf{traj}(\mathcal{S}) \neq \emptyset$. Then, for instance, the test

    $$\begin{array}{l}\sum\{\sigma'; \mathbf{pass} | \sigma' \in \mathbf{traj_u}(C_{\mathcal{S}})\} + \sum\{\sigma'; \mathbf{fail} | \sigma' \notin \mathbf{subtraj_u}(C_{\mathcal{S}})\} + \\ \sum\{o'; \mathcal{TC}_{o'} | o' \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \sum\{o'; \mathbf{fail} | o' \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma'; o; \mathbf{pass} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \in \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\} + \\ \sum\{\sigma'; o; \mathbf{pass} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \notin \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\}\end{array}$$

    with, for some $\sigma' \in \mathbf{traj}(C_{\mathcal{S}})$, $u = \sigma' \downarrow V_I$ and $j = u.ltime$ and $\mathcal{TC}_o = \mathcal{TC}$ and, for $o' \in (A_O \cap \mathbf{out}(C_{\mathcal{S}}))\backslash\{o\}$, $\mathcal{TC}_{o'} =$ **pass** will also lead to verdict **fail** for trace α.
  - Suppose that $\nu = \sigma$, with $\sigma \in \mathbf{traj}(C_{\mathcal{S}})$. Then, for instance, the test

    $$\begin{array}{l}\sum\{\sigma'; \mathcal{TC}_{\sigma'} | \sigma' \in \mathbf{traj_u}(C_{\mathcal{S}})\} + \sum\{\sigma'; \mathbf{fail} | \sigma \notin \mathbf{subtraj_u}(C_{\mathcal{S}})\} + \\ \sum\{o; \mathbf{pass} | o \in A_O \cap \mathbf{out}(C_{\mathcal{S}})\} + \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C_{\mathcal{S}})\} + \\ \sum\{\sigma'; o; \mathbf{pass} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \in \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\} + \\ \sum\{\sigma'; o; \mathbf{fail} | \sigma' \in \mathbf{subtraj_u}(C_{\mathcal{S}}) \wedge \sigma'.ltime < j \wedge \\ \qquad\qquad o \notin \mathbf{out}(C_{\mathcal{S}} \textbf{ after } \sigma')\}\end{array}$$

    with $u = \sigma \downarrow V_I$ and $j = \sigma.ltime$ and $\mathcal{TC}_\sigma = \mathcal{TC}$ and, for $\sigma' \in \mathbf{traj}(C_{\mathcal{S}})\backslash\{\sigma\}$, $\mathcal{TC}_{\sigma'} =$ **pass** will also lead to verdict **fail** for trace α.

4.7 Concluding Remarks

In this chapter we have defined an input-output conformance relation for hybrid systems. This relation defines in which cases we consider an hybrid implementation correct with respect to a hybrid specification. We have defined an inductive notion of test. We have proven that this notion of test is sound and exhaustive with respect to the hybrid input-output conformance relation.

The hybrid conformance relation and notion of test in this chapter have been defined in line with the input-output conformance theory by Tretmans [48] and the timed input-output conformance theories by Brandán-Briones and Brinksma [12], and by of Krichen and Tripakis [30], which we have described in Chapter 3. If we make a mapping between HTSs and TLTs, e.g. we define a TLTs as a HTSs with one output variable *time*, then if a timed implementation conforms to a timed specification according to Krichen and Tripakis, the hybrid equivalent of this implementation also conforms to the hybrid equivalent of the same specification and vice versa. For the same reasons as given by Krichen ant Tripakis [32] the previous statement does not hold for the relation between the conformance relation by Brandán-Briones and Brinksma and the hybrid conformance relation.

To stay in line with those conformance theories, we chose to let the test decides which continuous input to provide to the implementation. This makes our notion of test not suitable for testing implementations in which the continuous input depends on the observed output. E.g. systems that contain feedback loops. For a notion of test for these kind of systems the test has allow all trajectories from every state. The test execution mechanism has to adapt continuously and instantaneously to the observed input. We believe that this is practically infeasible.

Because in continuous systems the input behavior takes place in synchrony with the output behavior, we needed to explicitly define the restriction of the set of trajectories allowed by the implementation in the conformance relation. This filter on input trajectories made it necessary to introduce the notion of agile states. In a similar fashion to the quiescent action used by Tretmans, and Brandán-Briones and Brinksma, the agility symbol ξ allows us to conclude conformance between an implementation and a specification according to our intuition. However, in contrast to quiescence, we do not consider agility as an observable output. The reason is that in contrast to quiescence, agility can be observed from the implementation by means of applying and observing continuous output.

In order to test the applicability and the limitations of the notion of hybrid test in practice, we have developed a prototype test tool. In the next chapter, we first explain the test tools on which we have based our prototype tool and the language we use to make specifications for our hybrid test tool.

5

Hybrid χ and TorX

For our prototype hybrid test tool that we describe in Chapter 6, we use the same architecture and same way of operation as used in the discrete-event test tool TorX [7, 6]. For our prototype tool we use the χ language [5, 35] as specification language and the χ simulator tool set [25] to compute the set of transitions allowed by the specification after a trace.

We explain the discrete-event behavior of χ and the TorX tool by an example of testing a discrete-event χ simulation of the alternating bit protocol with TorX against a specification of the alternating bit protocol described in a dialect of PROMELA called TROJKA [55]. This research was published in [38].

At the end of this chapter we also discuss some other test tools and languages that we considered to use for our prototype test tool, namely: TTG [31, 29], timed-TorX [10], and Uppaal-TRON [33, 51], and the hybrid test tool Charon-tester [47, 46].

5.1 The Hybrid χ Language

The χ language [5, 35] is a hybrid process algebra with a hybrid transition system semantics (defined by SOS rules). In this section we give an overview of the language constructs relevant for this thesis.

A χ model has at the top level variable declarations, the parallel composition of continuous behavior described by (partial) differential equations and a number of χ processes. Variables are typed, e.g. as real numbers or strings. The χ language allows communication between processes through channels and shared variables. A process contains variable declarations, may contain continuous behavior described by (partial) differential equations, and may contain discrete-event behavior described by process terms. We describe these concepts more thoroughly by example in Section 5.2 and Section 5.3.

The following language constructs are relevant for this thesis:

- $|[\ldots]|$ denotes a scope;
- skip denotes internal activity;
- p; q denotes the alternative composition of process terms p and q;

- $v := e$ denotes the assignment of the value e to the variable v;
- $p \,[\!]\, q$ denotes the alternative composition of process terms p and q;
- $g \rightarrow p$ denotes the process that executes p provided that the guard g evaluates to $true$;
- $p \parallel q$ denotes the parallel composition of process terms p and q;
- $*(\ \ldots\ )$ denotes an unguarded repetition;
- $g \overset{*}{\rightarrow} p$ denotes the guarded repetition of process term p with guard g, which means process term p is executed as long as guard g evaluates to $true$;
- $h!e$ denotes a delayable send action of the value e over channel h;
- $h!!e$ denotes an undelayable send action of the value e over channel h;
- $h?e$ denotes a delayable receive action of the value e over channel h; and
- $h??e$ denotes an undelayable receive action of the value e over channel h.

Note that we use the more "user friendly" notation as described by Man and Schiffelers [35].

5.2 Modeling Discrete-event Behavior in χ

We use the well known alternating bit protocol as an example to illustrate how to model a system in χ. We deliberately introduce a mistake in the model presented in this section. In the next section we will explain how to test this model using TorX, and discuss how to fix the mistake in the model.

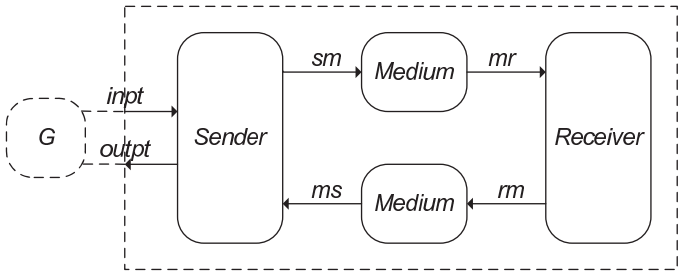

Figure 5.1: The Alternating Bit Protocol

The alternating bit protocol allows communication over lossy communication mediums. Lossy means a message might get corrupted or lost while inside the medium. A transmitter sends a message over a lossy medium to a receiver. If the message arrives uncorrupted, the receiver sends an acknowledgement over a second lossy

medium. If the message arrives corrupted the receiver sends an error message over the second medium. The acknowledgement or error message might get corrupted also. If the transmitter does not receive the proper acknowledgment, the original message is sent over the first medium again. This process is repeated until the transmitter receives the correct acknowledgment. After that, a new message is sent.

The protocol is depicted in Figure 5.1. The transmitter, mediums, and receiver are modeled as processes; process *Sender*, two instantiations of the process *Medium*, and process *Receiver*, respectively. The process G generates the messages and waits for the acknowledgements of successful transmission.

The Process *Sender* receives a message from G and as long as it does not receive an acknowledgement it keeps on transmitting it. Information is exchanged between processes via synchronous channels, in this case via input channel $inp?$ and output channel $outp!$. After the declaration of local variables msg and a, the body of this process is enclosed in an infinite repetition. A guarded repetition is used to keep on transmitting message as long as no acknowledgement $ack0$ or $ack1$ is received.

```
proc Sender( chan !sout, ?sin ,?inp, !outp : string ) =
[[ var msg : string = " ", a : string = " "
:: *( inp?msg
    ; ( a ∉ {"ack0", "ack1"} →* sout!msg; sin?a )
    ; outp!a
    )
]]
```

The Process *Receiver* receives a message uncorrupted ($msg0$ and $msg1$) or corrupted (c). This process uses a selection statement in which depending on the message, a particular acknowledgement ($ack0$, $ack1$ or $nack$) is selected.

```
proc Receiver( chan ?rin, !rout : string ) =
[[ var msg : string = " "
:: *( rin?msg;
      ( msg = "c" → rout!"nack"
      [] msg = "msg0" → rout!"ack0"
      [] msg = "msg1" → rout!"ack1"
      )
    )
]]
```

Note that process *Receiver* does not forwards messages to the environment after they are successfully received, which is common in the description of the alternating bit protocol.

The Process *Medium* is the communication medium between *Sender* and *Receiver*. Messages in the *Medium* may get corrupted, because the medium specifies a non-deterministic choice between sending the message msg or sending the corrupted message c.

```
proc Medium( chan ?cin, cout : string ) =
|[ var msg : string = " "
:: *( cin?msg; ( cout!msg [] cout!"c" ) )
]|
```

The Process G is used to generate messages to be sent by the process *Sender* to the process *Medium*. After the acknowledgement is received from the process *Sender* it generates another message (*msg1*). After the second acknowledgement the loop is repeated.

```
proc G( chan ?inp, !outp : string ) =
|[ var a : string
:: *( inp!"msg0"; outp?a; inp!"msg1"; outp?a )
]|
```

The processes need to be instantiated in a model. In this model the connections between processes are specified. All processes are executed in parallel.

```
model abp( )
|[ chan sm, ms, mr, rm, inpt, outpt : string
:: G(inpt, outpt) || Sender(sm, ms, inpt, outpt) || Medium(sm, mr)
|| Medium(rm, ms) || Receiver(mr, rm)
]|
```

After the processes have been instantiated, the χ-model can be compiled and executed. For this model, the χ compiler does not find an error and an executable simulation is created. However, that does not mean the model correctly implements the alternating bit protocol.

5.3 Modeling Continuous Behavior in χ

Hybrid χ is an extension of the discrete-event part of χ (explained in the previous section) (or, more accurately, a discrete-event χ model is a hybrid χ model without continuous behavior).

In χ, continuous behavior is modeled by continuous variables. The language χ leaves open in what form the continuous behavior is defined. However, in practice, the continuous behavior is usually defined by equations on variables and differential equations. At the time of writing of this thesis, the χ simulator tool set only allows equations and standard differential equations to specify continuous behavior.

Figure 5.2 depicts a system with continuous behavior. Each block with rounded corners depicts a χ process. The arrows depict the communication channels between processes and their direction. In this system, the thermostat observes the temperature of a room and sends messages to a heater over a medium to either turn the heater on or turn the heater off. The thermostat and the room share continuous

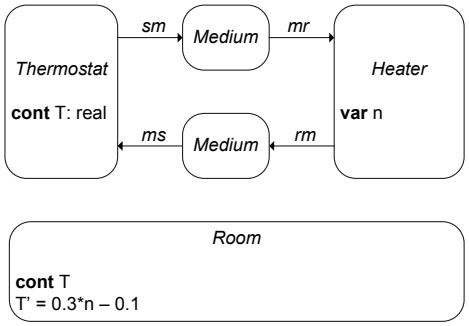

Figure 5.2: A Hybrid Thermostat Model

variable T and the heater and the room share the discrete variable n. The heater sends back an acknowledgement if it has correctly received a message to turn on or to turn off, after which the thermostat continues to observe the temperature of the room. If the heater is turned on, then the room temperature increases and if the heater is turned off the room temperature decreases.

The process *Thermostat* is similar to the process *Sender* in the previous section. It shares the continuous variable T, which is defined as a parameter of the process, with the process *Room*. In χ, the flow of continuous variables is specified in one process or inside the model. The process *Thermostat* does not influence the room temperature. Hence, the thermostat does not define the flow of T. The thermostat waits until the temperate T drops either below 15 ^oC or exceeds 20 ^oC after which it turns on or turns off the heater respectively. This message is sent to the heater via the medium until an acknowledgement from the heater is received over the medium. After an acknowledgement is received, the whole process is repeated.

```
proc Thermostat( cont : T : real, chan !sout, ?sin : string ) =
|[ var msg : string = " ", a : string = " "
:: *(  ( T ≤ 15.0 → msg := "turn heater on"
       [] T ≥ 20.0 → msg := "turn heater off"
       )
     ; sout!msg; sin?a
     ; ( a ∉ {"ack0", "ack1"} →* sout!msg; sin?a )
   )
]|
```

The process *Room* only models the temperature flow of the room. This is modeled by T' which models the first derivative of variable T. If the shared variable n is set to 1.0 by the heater, then the room temperature increases with $0.3 * 1.0 - 0.1 = 0.2\ ^oC/sec$. If the shared variable n is set to 0.0 by the heater, then the room temperature decreases with $0.3 * 0.0 - 0.1 = -0.1\ ^oC/sec$.

```
proc Room( cont T : real, var n : real ) =
|[ T' = 0.3 * n - 0.1
]|
```

The process *Heater* is similar to the process *Receiver* in the previous section. If it receives a message to turn on, it sets the shared discrete variable n to 1.0, which causes the room temperature T to increase (as described below). If it receives a message to turn off, it sets the shared discrete variable n to 0.0, which causes the room temperature T to decrease.

```
proc Heater( var n : real, chan ?rin, !rout : string ) =
|[ var msg : string = " "
:: *( rin?msg;
      ( msg = "c" -> rout!"nack"
      [] msg = "turn heater on" -> rout!"ack0"; n := 1.0
      [] msg = "turn heater off" -> rout!"ack1"; n := 0.0
      )
    )
]|
```

The processes *Thermostat*, *Room* and *Heater* are composed together with two instantiations of the process *Medium* as described in the previous section, into one model. The model also initializes the variables T and n.

```
model RoomThermos( )
|[ cont T : real = 17.5, var n : real = 0.0
  chan sm, ms, mr, rm : string
:: Thermostat(T, sm, ms) || Medium(sm, mr) ||
   Medium(rm, ms) || Heater(n, mr, rm) || Room(T, n)
]|
```

5.4 The Hybrid χ Simulator

The hybrid χ tool set [25] allows simulation of discrete-event, timed, as well as hybrid χ models. A simulation run starts with selecting, non deterministically or by the user, an action or a duration. The simulator computes the trajectories of continuous variables and one new state that is reached by performing an action or that is reached by waiting for the selected duration. From the set of actions and durations allowed by this state a new action or duration is selected, after which a new reachable state is computed. This process continuous until the user terminates the test or until the χ simulator reaches a deadlock. During a simulation run, the continuous behavior of the simulation run is plotted and the actions are displayed on screen.

Underneath the χ simulator lies a class library that makes it possible to compute the actions allowed in a state, to synchronize processes through channel communications, to compute the trajectories allowed in a state, and to compute the reachable states after performing an action or trajectory. Differential equations are can be either solved by using a symbolic equation solver or a numerical equation solver. This set of classes is called the stepper. It contains all the methods we would require to implement a test generation and execution algorithm for hybrid tests.

5.5 TorX

The TorX test tool was developed within the Côte-de-Resyste project [16], a collaboration between the University of Twente, Eindhoven University, Philips Research Laboratories and Lucent Technologies. It can be used for testing hardware and software.

Several case-studies have been performed with TorX (e.g. a highway tolling system [54] and a communication protocol [21]). The version of TorX that we discuss below implements the notion of test of Tretmans as defined in Chapter 3.

The TorX tool supports testing of discrete event systems according to this conformance relation. An input generated from a specification is applied to the system under test, and the output from this system is compared to the specified output. When it is possible to perform the observed output transition in the specification, the test is passed. The architecture is depicted in Figure 5.3.

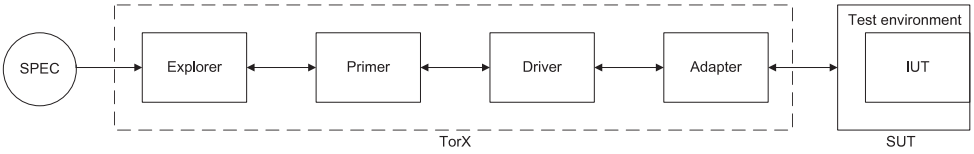

Figure 5.3: Torx Architecture

SPEC: The formal model, from which the test-cases are generated. TorX is able to extract test cases from several specification languages, e.g. TROJKA [55] (a dialect of the PROMELA language) and LOTOS. It is also possible to implement an automaton or labeled transition system in C and generate test cases from these models.

Explorer: This component offers functionality to explore the transition graph of a specification. It is language specific. For a given state it provides the set of transitions (input and output) from that state. E.g. an explorer is available that is based on the *OPEN/CAESAR* [19] interface, and one that is based on the *SPIN* [26] interface.

Primer: This component implements the test derivation algorithm. It also provides functionality to offer input stimuli to the implementation under test and to check output observations from the implementation under test. Test selection is currently done at random (with random seeds) or using test purposes [53].

Driver: This component controls the progress of the testing process. It decides whether to do an input action or to observe possible output from the implementation. It uses the primer to select an input action from the specification and the adapter to observe the actual output from the implementation.

Adapter: This component is the connection between the test tool and the system under test. It is responsible for translating input transitions from the specification to readable inputs for the implementation, and for translating outputs received from the implementation back to output actions. These output actions are then matched to the output transitions in the specification.

SUT: The System Under Test (SUT) consists of the actual implementation under test (IUT), together with the test environment (e.g. drivers, stubs, hardware and operating system). An input from the adapter is passed on to the implementation under test. The output from the SUT is communicated back to the adapter.

The initial purpose of the tool was to test real implementations automatically. In the rest of this paper we show it can also be used to test models. This can be useful if we want to validate different versions of a model or if we want to validate parts of a complex model.

5.6 Testing Discrete χ models with TorX

The simulation model presented in the previous section contains a mistake that might not be spotted immediately. Usually, only when the simulation model appears to run incorrectly a designer starts debugging the model. He might add extra debug code to output additional information on the state of the simulation at particular points. He might study the code itself, which is time consuming and does not guarantee success either.

We propose to use TorX for automated model-based testing of this model. To be able to test a simulation model, four steps have to be taken:

1. The model for generating test-cases needs to be built if this model does not exist already. Once this model has been made it can be re-used or adapted for any simulation model with similar behavior.

2. Both models, the simulation model and the model used for test generation need to be *open*, i.e. external channels need to be observable for the test tool.

3. The connection between the test tool and the simulator needs to be built. An adapter needs to be implemented which can be used in testing χ models. We have adapted an existing adapter, written in Python, for this purpose.

4. Within the adapter a translation scheme needs to be made for syntactic differences between the model under test and the model that serves as the specification.

TorX is able to derive test-cases from a dialect of the PROMELA language called TROJKA. The only difference between PROMELA and TROJKA is that in the TROJKA-language a channel can be defined as observable. This means that the communication over this channel is visible (e.g. for TorX to use as test-input or output). The messages over channels that are not declared observable are not used for test-case generation.

The correct behavior of the alternating bit protocol is that for every message sent, eventually the corresponding acknowledgement should be received. Only after an acknowledgment has been received the next message can be sent. The behavior of the system can be specified in TROJKA as follows:

```
mtype = {call, result, send,  m0, m1, a0, a1};

chan TDRV__channel = [0] of { mtype,mtype,mtype } OBSERVABLE;

active proctype Sender() {
      do
      :: TDRV__channel?call,send,m0;
         TDRV__channel!result,send,a0;
         TDRV__channel?call,send,m1;
         TDRV__channel!result,send,a1
      od
}
```

In this model input message `m0` is sent first, then output message `a0` is received. Then `m1` is sent after which `a1` is received. After that, `m0` is sent again and `a0` is received and so on. A simulation model is normally *closed*. There is no interaction between the simulation (processes) and "the outside world" (e.g. other tools or a user) to guide the simulation. In order to test the χ-model of the alternating bit protocol interaction with the TorX tool is needed. The TROJKA-model is open by means of the observable channel. The χ model can be made open by replacing process G (see Section 3) with a new process *IO*. Replacing process G does not affect the behavior of the alternating bit protocol itself (which is implemented by the processes *Sender*, *Medium*, and *Receiver*).

Instead of alternating between $msg0$ and $msg1$ inside a process, the message to be sent is received from a special purpose channel. In the χ-language this is a channel without a name and it is normally used for receiving input from a unix-terminal

```
proc io( chan inp : string, outp : string ) =
|[ var m : string
| *( ?m; inp!m; outp?m; !m )
]|
```

command line, or for printing output to the command line. For our purpose we need TorX to communicate with this special channel.

We have built an adapter to establish the connection between TorX and the simulation through this channel. It consists of a generic part and a model specific part. The generic part can be re-used; it implements a connection between TorX and χ by means of pipes. Because a pipe mechanism is used every data type used in the TROJKA-model needs to be translated to a string. The model-specific part is the translation of input and output messages between the TROJKA-model and the χ-model. PROMELA, and therefore TROJKA, allows data types name, bit, and byte. The χ language also allows data types string, boolean, and natural. To make testing possible, we need to define a mapping of messages defined in the TROJKA-model to the messages defined in the χ model, and the other way around. In our alternating bit protocol example the message `m0` in the TROJKA-model is the message $msg0$ in the χ model. The same holds for the other message `m1` and the acknowledgments. In the adapter we define a mapping between these messages.

Our adapter presupposes that the TROJKA-model meets some mild syntactic conventions (that were already used in the TROJKA model earlier in this section):

- The channel names have to start with `TDRV__` in order to use the correct Python class in which the adapter has been implemented.
- The first part of the message has to be `call` or `result` to indicate an input to the simulation or an output from the simulation respectively.
- The second part of the message has to be the function call in the adapter (`send`) which sends the input to the simulation and returns the output from the simulation.

Using the adapter we can now test the χ-simulation of the alternating bit protocol with TorX. During test execution TorX generates a message sequence chart (MSC) [13]. The MSC shows the components (e.g. processes) involved and the interaction between those components. In this case the components are the test driver `tdrv`, the implementation `iut` and the outside world `(out)`. The interaction between components is indicated by arrows. The order in which the messages take place is from top to bottom. The top message takes place first. The distance between two messages is not an indication for the amount of time that passed between those messages.

The result of testing our alternating bit protocol is depicted in Figure 5.4.

First, the message `m0` is provided to the implementation and correctly acknowledged by the implementation. Then, the implementation cannot produce an acknowledge

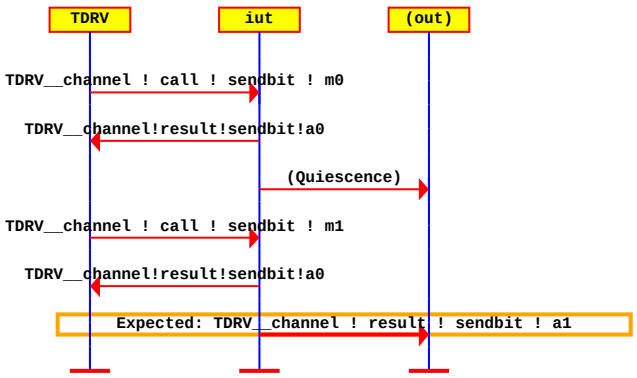

Figure 5.4: Error Found in Model-based Testing the Alternating Bit Protocol

output without an input message and therefore it produces a quiescence message. After providing message `m1` as input, acknowledgement `a0` is received. However, acknowledgement `a1` was expected as output. This was the acknowledgement to follow message `m1` in the TROJKA-model. Repeating this test with `m1` as first input shows that upon message `m0`, `a1` is received. In this case acknowledgement `a0` was the expected output from the simulation.

These observations lead to the conclusion a mistake could have been made in process *Sender*: When the first acknowledgement is received the variable a is assigned that value; after sending the second message a still has this value; therefore $a \in \{"ack0","ack1"\}$ and value a is returned as output. This leads to the conclusion that the second time the message is never sent over the medium. The solution for this problem is to read the first response on the current message before evaluating the guard of the repetition for the first time:

$$
\begin{array}{l}
\mathsf{proc}\ Sender(\ \mathsf{chan}\ !sout,\ ?sin\ ,?inp,\ !outp : \mathsf{string}\ )\ = \\
[\![\ \mathsf{var}\ msg : \mathsf{string} = \text{“ ”}, a : \mathsf{string} = \text{“ ”} \\
::\ *(\ inp?msg;\ sout!msg;\ sin?a \\
\qquad ;\ (\ a \notin \{\text{“}ack0\text{”}, \text{“}ack1\text{”}\}\ \xrightarrow{*}\ sout!msg; sin?a\ ) \\
\qquad ;\ outp!a \\
\qquad) \\
]\!]
\end{array}
$$

Although this was a small example, it contained a mistake. By examining the model it can be hard to spot the mistake. With the help of model-based testing the mistake was found and the model used for validating the simulation model only contained ten lines of code.

5.7 Other Test Tools

A timed version of TorX was implemented by Bohnenkamp and Belinfante [10] . Timed TorX was developed as part of the TANGRAM project. They also redefined timed input-output conformance with quiescence for timed automata (instead of timed transition system). The two theories are the same except they do not lay a time bound M on when to conclude quiescence. They leave that up to the implementation. Timed-TorX is an extension of TorX. An input can be selected together with the time it has to be applied to the implementation. If an output action is observed from the implementation it is validated against the test whether this output was allowed. If the output was not allowed, the verdict fail is concluded. Test generation and execution is performed on-the-fly, like in TorX. Some case studies were performed with timed-TorX. First, some timed automata implementations (implemented in C) were tested against some timed automata specifications. In this case, tests were not executed in real-time. The automaton produced the output together with the "virtual" time they were performed. Second, an experiment was conducted with testing timed properties of a software controller of a waferstepper machine. As a result some timing mistakes were found in that component.

Krichen and Tripakis implemented the prototype timed test tool TTG [29]. It was built upon the IF environment [11, 52]. They performed some toy example case studies and they performed a case study on the K9 rover [31].

Larsen [33] et al. also implemented a timed test tool named UPPAAL-TRON [51]. This test tool is based upon the UPPAAL model-checking toolkit. They did not define their own conformance relation (and formal test generation procedure). Their works seems to be based on Krichen and Tripakis's theory, or they have been influencing eachother. UPPAAL-TRON does not use a quiescence action either. In [33], Larsen et al. describe a case study in which they have used UPPAAL-TRON in an industrial case-study on testing a controller for industrial cooling devices.

The only hybrid model-based test tool to our knowledge is the Charon tester [47, 46]. This is a prototype tool implemented in the Charon framework [2]. The Charon tester takes a different approach than those of the other test tools mentioned above. The Charon language is a hierarchical graphical modeling language based on the theory of hybrid input-output automata. The Charon tool-set is used for hybrid simulation and runtime verification. In runtime verification a small program is generated from a property that is executed together with the implementation. If the property is violated this is reported. With the Charon tester, besides properties, also an environment (of the implementation) is modeled. A test generator is implemented that is also executed together with the implementation and that selects input for the implementation from the environment model. If a property is violated the test fails. The advantage of this approach is that inaccuracy in observations is not an issue, e.g. there is no clock skew because both test and implementation run on the same platform. The disadvantage of this approach is that it is not flexible. For every different environment, different test generator, and different execution platform, new code needs to written and generated, and embedded in the system under test.

Besides hybrid χ we also considered other hybrid languages as potential specification language and other tools as basis for a prototype tool, namely haron [2], PHAver [18], and HyTech [24]. In the end we have chosen χ. The first reason is that χ has been developed at the Technische Universiteit Eindhoven at the Mechanical Engineering Department, and therefore developers are close by for assistance. The second reason is that the χ simulator has been built such that it is possible to implement a test tool application reusing underlying class libraries, without needing to adapt them.

6

Test tool Implementation

This chapter describes the issues involved in developing a test tool that is based on the theory for hybrid testing introduced in Chapter 4. This chapter explains a number of design issues and implementation issues considering the development of a hybrid conformance test tool in general. Design issues we call those issues that are related to the design of the test algorithm. Implementation issues we call those issues that are related to practical circumstances. For instance, how to select input is a design issue while how to execute the tests in real-time is an implementation issue.

We have implemented a prototype version of a test tool for hybrid systems. The remainder of this chapter describes two examples that show that the prototype tool works and is able to find mistakes in an implementation. In Chapter 7 we present an industrial case study that we have performed with this tool.

6.1 Tool Architecture

All conformance test tools discussed in Chapter 5 (TorX [7], TTG [29], and Uppaal-TRON [33]) use the same basic architecture. Since our notion of test is based on the test definitions that are behind these tools, we choose the same architecture when implementing a hybrid test tool.

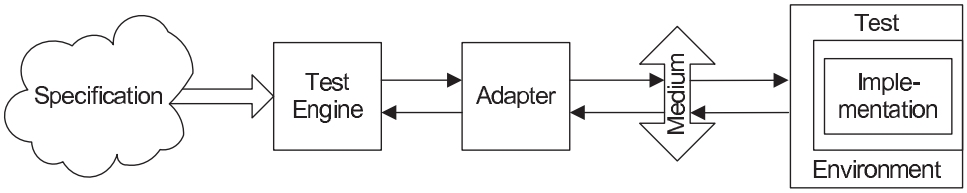

Figure 6.1: Test Tool Architecture

The test engine implements the test generation procedure. It steps through the specification and computes the sets of allowed input actions, output actions, and

trajectories. It selects an input action or trajectory on input variables to be applied to the implementation. If an output action or continuous output is observed, then it evaluates whether this output is allowed by the specification. If some output is observed that is not allowed according to the specification, then the test is terminated with the verdict fail. As long as the verdict fail is not given, the test can be terminated by some stop criterion. In this case the test terminates with the verdict pass.

The adapter transforms input to a format that is suitable to be sent through the communication medium to the implementation (through the test environment). It also transforms output received over the communication medium from the implementation back to a format suitable for comparison with the specified output.

In the architecture, medium refers to all hardware and software for the communication between the test tool and the implementation. The medium can for instance be an ordinary TCP/IP network but also dedicated busses, or wires. Due to the presence of the medium, the test tool and implementation do not need to be on the same execution platform. The platform on which the test engine and adapter are executed is usually called the *specification side* of the test setup and the platform on which the implementation and test environment are executed is usually called the *implementation side* of the test setup.

The test environment handles the communication over the medium from the implementation side. This can for instance be packing and unpacking data packages sent over the medium. It is also used for instance to add time-stamps to the output.

Before we can build a test tool we need to design a test generation and execution algorithm. There are two major decisions to take. First we need to decide whether to generate the test before the execution or to generate the test at runtime *on-the-fly*. Second, we need to decide whether we apply real continuous input and observe real continuous output, or only use samples of continuous input and observe samples of continuous output.

6.2 Test Generation and Execution

Recall that a test is viewed as a tree shaped HTS with transitions labeled with input actions, output actions, and trajectories, and with leaf states labeled with verdicts **pass** or **fail**. The notion of test in Chapter 4, does not define when to select and apply input actions, when to observe output actions, when to select continuous input, when to apply continuous input to the implementation and when to observe continuous output from the implementation.

In order to implement hybrid model-based testing we need to decide whether to generate a test from the specification before the execution or to generate the test from the specification during the execution. Test generation at runtime is called *on-the-fly* test generation and execution. The advantage of generating a test before the execution is that less computation time is needed at runtime than in the case the tests are generated at runtime. The advantage of generating test during the

execution is that as a whole it is faster compared to test generation before execution. If we generate tests on-the-fly then we only need to step through the specification, while if we generate the test before the execution, then we first need to step through the specification for generating the test and then we need to step through the test in order to execute it. We discuss the options and make a choice.

A recursive algorithm for generating tests before execution is derived directly from the notion of test described in Chapter 4. This algorithm is shown in Figure 6.2.

---

algorithm tcg(C) =

select either

1. return **pass**
2. compute $I = \{a | a \in A_I \wedge C \textbf{ after } a \neq \emptyset\}$
   if $I \neq \emptyset$ then select an $i \in I$
   return $i; \textbf{tcg}(C \textbf{ after } i)$
3. if $\textbf{traj}(C) = \emptyset$ then
   return

$$\sum\{o; \textbf{tcg}(C \textbf{ after } o) | o \in A_o \cap \textbf{out}(C)\} +$$
$$\sum\{o; \textbf{fail} | o \in A_o \backslash \textbf{out}(C)\} +$$
$$\sum\{\sigma; \textbf{fail} | \sigma \in \Sigma_{\mathcal{S}}\}$$

4. if $\textbf{traj}(C) \neq \emptyset$ then
   select an $u \in \{\sigma \downarrow V_I | \sigma \in \textbf{traj}(C)\}$
   compute $\textbf{traj}_{\textbf{u}}(C) = \{\sigma | \sigma \downarrow V_I = u \wedge \sigma \in \textbf{traj}(C)\}$
   compute $\textbf{subtraj}_{\textbf{u}}(C) = \{\sigma | \exists_{\sigma' \in \textbf{traj}_{\textbf{u}}(C)} : \sigma \leq \sigma'\}$
   return

$$\sum\{\sigma; \textbf{tcg}(C \textbf{ after } \sigma) | \sigma \in \textbf{traj}_{\textbf{u}}(C)\} +$$
$$\sum\{\sigma; \textbf{fail} | \sigma \notin \textbf{subtraj}_{\textbf{u}}(C)\} +$$
$$\sum\{o; \textbf{tcg}(C \textbf{ after } o) | o \in A_O \cap \textbf{out}(C)\} +$$
$$\sum\{o; \textbf{fail} | o \in A_O \backslash \textbf{out}(C)\} +$$
$$\sum\{\sigma'; o; \textbf{tcg}(C \textbf{ after } \sigma' o) | \sigma' \in \textbf{subtraj}_{\textbf{u}}(C) \wedge$$
$$\sigma'.ltime < u.ltime \ \wedge \ o \in A_O \cap \textbf{out}(C \textbf{ after } \sigma')\} +$$
$$\sum\{\sigma'; o; \textbf{fail} | \sigma' \in \textbf{subtraj}_{\textbf{u}}(C) \wedge$$
$$\sigma'.ltime < u.ltime \ \wedge o \in A_O \setminus \textbf{out}(C \textbf{ after } \sigma')\}$$

Figure 6.2: A Recursive Hybrid Test Algorithm

For this algorithm to terminate we eventually need to select the verdict **pass** for every leaf state that is not the verdict **fail**.

However, the resulting test tree becomes infinite as soon as step 3 or step 4 is applied. Therefore in order to make test generation before execution implementable and usable in practice, we need a higher level representation for tests. In essence, a test is a partial specification with verdicts added. It might be possible to describe a

test as a hybrid automaton with failure states added. In such automaton number of trajectories is grouped into one mode. However, a test automaton may grow to the size of the specification. Furthermore, a new test needs to be generated for every test with (at least one) different input action or trajectory compared to the tests generated previously. Therefore, test generation before execution is not considered a viable option.

The solution is to simultaneously generate and execute a test directly from the specification. In this case every step of the recursive test generation algorithm is performed on the implementation, before the next recursive step is taken or the test is terminated. Note that in the algorithm described below we still use mathematical notion and possibly infinite sets. In practice we can only handle finite representations of sets of actions and sets of trajectories. We describe this issue in further detail in Section 6.4.1.

Let $\mathcal{S}$ be a specification and let $C \subseteq S$ be a set of states of $\mathcal{S}$. On-the-fly test generation and execution is implemented by the recursive algorithm **otftcg**(C) which repeatedly selects one of the following 4 steps:

1. terminate testing with verdict **pass**.

2. select an input action i, such that C **after** $i \neq \emptyset$, and apply i to the implementation and continue testing with **otftcg**$(C$ **after** $i)$

3. if **traj**$(C) = \emptyset$ then select an arbitrary $\sigma \in \Sigma$ with arbitrary duration and start applying $\sigma \downarrow V_I$ to the implementation:

   (a) if an output action o is observed immediately from the implementation and $o \in A_O \cap$ **out**(C) then continue testing with **otftcg**$(C$ **after** $o)$;

   (b) if an output action o is observed immediately from the implementation but $o \in A_O \setminus$ **out**(C) then terminate testing with verdict **fail**; and

   (c) if no output action is observed from the implementation then terminate testing with the verdict **fail**;

4. if **traj**$(C) \neq \emptyset$ then select a $\sigma \in$ **traj**(C) with a duration t, apply $\sigma \downarrow V_I$ to the implementation and observe the continuous output of the implementation, denoted by a trajectory σ_{V_O} on variables V_O:

   (a) if an output action o is observed immediately from the implementation and $o \in A_O \cap$ **out**(C), then continue testing with **otftcg**$(C$ **after** $o)$;

   (b) if an output action o is observed immediately from the implementation but $o \in A_O \setminus$ **out**(C), then terminate testing with verdict **fail**;

   (c) if at time $t' < t$, the trajectory $\sigma_{V_I} \leq \sigma \downarrow V_I$ with $\sigma_{V_I}.ltime = t'$ has been applied to the implementation and σ_{V_O} has been observed from the implementation but there does not exist a $\sigma \in$ **traj**(C) such that $\sigma \downarrow V_I = \sigma_{V_I}$ and $\sigma \downarrow V_O = \sigma_{V_O}$, then terminate testing with verdict fail;

(d) if at time $t' < t$ an output action o is observed and the trajectory $\sigma_{V_I} \leq \sigma \downarrow V_I$ with $\sigma_{V_I}.ltime = t'$ has been applied to the implementation and the trajectory σ_{V_O} has been observed from the implementation and there does exist a $\sigma \in \mathbf{traj}(C)$ such that $\sigma \downarrow V_I = \sigma_{V_I}$ and $\sigma \downarrow V_O = \sigma_{V_O}$, denoted by $\sigma_{V_I \cup V_O}$, then
   i. if $o \in \mathbf{out}(C \textbf{ after } A_O \cap \sigma_{V_I \cup V_O})$, then continue testing with $\mathbf{otftcg}(C \textbf{ after } o)$;
   ii. if $o \in A_O \setminus \mathbf{out}(C \textbf{ after } \sigma_{V_I \cup V_O})$, then terminate testing with verdict **fail**;

(e) if at time t the trajectory σ_{V_O} is observed from the implementation and there exists a $\sigma' \in \mathbf{traj}(C)$ such that $\sigma' \downarrow V_I = \sigma \downarrow V_I$ and $\sigma' \downarrow V_O = \sigma_{V_O}$ then continue testing with $\mathbf{otftcg}(C \textbf{ after } \sigma')$;

Then, $\mathbf{otftcg}(\{s_o\})$ performs the on-the-fly test generation and execution.

In on-the-fly test generation, the test tree does not need to be generated any further for the output that is not observed at runtime. Even more, once a test step has been generated and executed, that part of the test tree does not need to be stored. Furthermore, with test generation before execution we first need to step through the specification to generate the test and then we need to step through the test to execute the test on the implementation, while with on-the-fly test generation and execution we only need to step through the specification once and a smaller part of it.

6.3 Sampling

To test with continuous behavior we need to solve two problems. The first problem is how to accurately stimulate a hybrid system with continuous input and how to accurately observe continuous output from a hybrid system. An actuator, e.g. a heater, as part of the medium to stimulate a sensor of an implementation cannot be controlled accurately enough to exactly produce the input as prescribed by the test. A sensor as part of the medium to observe physical behavior of the implementation cannot observe accurately enough because influences from the environment of the implementation cannot be controlled. The second problem is that, even if the continuous behavior can be accurately controlled, e.g. with electronic signals, time is needed to select and apply continuous input and to observe and validate continuous output. Therefore, selecting input and validating output has to take place while new input is applied and new output is observed.

By using sampled behavior instead of real continuous behavior the sample rate can be adjusted such that there is time to compute new input samples and evaluate output samples in between sending and receiving samples.

Sampling often suffices for the purpose of the test engineer e.g. in the case that:

- a controller is tested the controller might already sample the continuous behavior observed through a sensor; or

- the test engineer has enough confidence that a well chosen sample rate sufficiently characterizes the continuous behavior of the implementation.

What is the best sample rate depends on the purpose of the test and on the implementation. Sometimes a static sample rate suffices. E.g. if we test a software controller with a sample rate of 1 second, then a static sample rate of 1 second is sufficient. Sometimes a dynamic sample rate might be advantageous. In this case the sample rate is dynamically changed during the test execution. E.g. the test tool generates more samples depending on the slope of the trajectory.

6.3.1 Sampled Hybrid Input-output Conformance

We will formalize the use of sampled continuous behavior in a new hybrid conformance relation and notion of test. In this relation the implementation and specification remain hybrid entities. The difference with the conformance relation defined in the previous chapter is that sampled trajectories are used.

With a fixed sampling rate, the first sample point of a trajectory can only be determined based on the duration of the preceding trace. E.g. if a trajectory lasts 3.5 seconds and if the sample rate is 1 second, starting from time 0, then the last sample of this trajectory is taken at 3 seconds and the first sample of the next trajectory has to be taken at 0.5 seconds. Recall that every trajectory is defined over a left-open interval starting at 0.

In order to define sampling, we first need to define the duration of a sequence of actions and trajectories.

Definition 6.1. *Let $\alpha, \alpha' \in (A \cup \Sigma)$. Let $a \in A$ and let $\sigma \in \Sigma$. Then the* duration *of α, denoted by $\alpha.ltime$ is defined as:*

- *if $\alpha = \epsilon$ then $\alpha.ltime = 0$;*
- *if $\alpha = a\alpha'$ then $\alpha.ltime = \alpha'.ltime$; and*
- *if $\alpha = \sigma\alpha'$ then $\alpha.ltime = \sigma.ltime + \alpha'.ltime$.*

The first sample point of a trajectories is computed as follows. Let sr be the sample rate. Let $\alpha.ltime$ be the duration of sequence α and let **mod** be the standard mathematical modulo function, recursively defined for positive numbers a and b (including 0) by:

$$\mathbf{mod}(a, b) = \begin{cases} a & \text{, if } a < b \\ \mathbf{mod}(a - b, b) & \text{, otherwise.} \end{cases}$$

Then the first sample point for trajectory σ in the trace $\alpha\sigma$ is $sr - \mathbf{mod}(\alpha.ltime, sr)$. With this first sample point we define the samples of a trajectory.

Definition 6.2. *Let sr be the sample rate. Let σ be a trajectory. Let $0 < p \leq sr$ be the first sample point of the trajectory and let $\perp$ denote a special value not in*

$\mathbb{R}$. *Then,* the sampled trajectory of σ, *denoted by* $\mathbf{sampled}_{sr}(\sigma, p)$, *is a trajectory with, for all* $t \in (0, \sigma.ltime]$*:*

$$\mathbf{sampled}_{sr}(\sigma,p)(t) = \begin{cases} \sigma(t) \text{ , } \textit{if } \mathbf{mod}(sr + t - p, sr) = 0 \\ \perp \quad \text{ , } \textit{otherwise.} \end{cases}$$

For a set of trajectories Σ *we define:*

$$\mathbf{sampled}_{sr}(\Sigma,p) = \bigcup_{\sigma\in\Sigma} \mathbf{sampled}_{sr}(\sigma,p).$$

For a trace $\alpha \in (A \cup \Sigma)^*$ *we define:*

- *if* $\alpha = \epsilon$*, then*
$$\mathbf{sampled}_{sr}(\alpha,p) = \epsilon;$$
- *if* $\alpha = a\alpha'$*, with* $a \in A$ *and* $\alpha' \in (A \cup \Sigma)^*$*, then*
$$\mathbf{sampled}_{sr}(\alpha,p) = a\ \mathbf{sampled}_{sr}(\alpha,p); \textit{and}$$
- *if* $\alpha = \sigma\alpha'$*, with* $\sigma \in \Sigma$ *and* $\alpha' \in (A \cup \Sigma)^*$*, and* $np = sr - \mathbf{mod}(sr + \sigma.ltime - p, sr)$ *then*
$$\mathbf{sampled}_{sr}(\alpha,p) = \mathbf{sampled}_{sr}(\sigma,p)\ \mathbf{sampled}_{sr}(\alpha,np).$$

When sampling is applied to the trajectories, the following may happen. Consider the two trajectories σ_1 and σ_2 depicted in Figure 6.3.

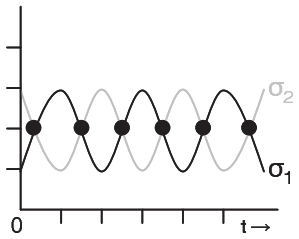

Figure 6.3: Two Trajectories with the same Sample Points

With a certain sample rate, both σ_1 and σ_2 have the same value. It is clear that the samples do not characterize a unique trajectory. Upon receiving these samples as input, an implementation may therefore react as if it received σ_1 as input or as if it received σ_2 as input. The specification and implementation may also contain different traces which have the same valuations when sampling is applied.

Definition 6.3. *For a set of traces* $\Omega \subseteq (A \cup \Sigma)^*$ *we define:*

$$C\ \mathbf{after}\ \Omega = \bigcup_{\alpha\in\Omega} C\ \mathbf{after}\ \alpha.$$

Using Definitions 4.3, 4.5, 4.8, 4.12, and 6.3 we redefine the hybrid conformance relation for sampled continuous behavior.

Definition 6.4. *Let $\mathcal{S}$ be a specification and let $\mathcal{I}$ be an input-enabled implementation. Let sr be the sample rate. Let, for all traces $\alpha \in traces(\mathcal{S})$, $p_\alpha = sr - \mathbf{mod}(\alpha.ltime, sr)$ be the next sample point after the trace α. Let $\Omega_\alpha = \{\omega \in traces(\mathcal{S}) | \mathbf{sampled}_{sr}(\omega, 0) = \mathbf{sampled}_{sr}(\alpha, 0)\}$ be the set of all traces with after sampling the same valuations as the sampled trace α. We say that $\mathcal{I}$ is sampled hybrid input-output conform $\mathcal{S}$, denoted by $\mathcal{I}$ $\mathbf{hioco}_{sr}$ $\mathcal{S}$, iff for all traces $\alpha \in traces(\mathcal{S})$:*

$$\mathbf{out}(\mathcal{I} \textbf{ after } \Omega_\alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \Omega_\alpha) \wedge$$

$$\mathbf{infilter}(\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{I} \textbf{ after } \Omega_\alpha), p_\alpha), \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha)) \subseteq$$

$$\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha).$$

6.3.2 Sampled Hybrid Tests

For the notion of test without sampling we needed the set of all trajectories with the same trajectory on input variables. For a notion of test with sampling we need the set of all trajectories, with the same sampled trajectory on input variables.

Definition 6.5. *Let $\mathcal{S} = (S, s_0, A_\tau \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a specification with continuous variables $V = V_I \cup V_O$ and let $C \subseteq S$ be a non-empty set of states. Let sr be the sample rate and let $0 < p \leq sr$ be the first sample point of the next trajectory; then the set of hybrid tests with sample rate sr, denoted by $hTests_{sr}(C, p)$ is inductively defined as follows:*

1. ***pass*** *is an element of $hTests_{sr}(C, p)$.*
2. *Suppose $i \in A_I$ and C **after** $i \neq \emptyset$ and let $\mathcal{TC}' \in hTests_{sr}(C \textbf{ after } i, p)$, then $i; \mathcal{TC}'$ is an element of $hTests_{sr}(C, p)$.*
3. *Suppose $\mathbf{traj}(C) = \emptyset$ and let, for all $o \in A_O \cap \mathbf{out}(C)$, $\mathcal{TC}_o \in hTests_{sr}(C \textbf{ after } o, p)$, then*

$$\begin{array}{l} \sum\{o; \mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C)\} + \\ \sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C)\} + \\ \sum\{\sigma; \mathbf{fail} | \sigma \in \mathbf{sampled}_{sr}(\Sigma, p)\} \end{array}$$

   is an element of $hTests_{sr}(C, p)$.

4. *Denote by $\Sigma_\sigma = \{\sigma' | \mathbf{sampled}_{sr}(\sigma', p) = \sigma\}$ the set of trajectories with after sampling the same valuations as the sampled trajectory σ. Denote by $u \in \{\sigma \downarrow V_I | \sigma \in \mathbf{sampled}_{sr}(\mathbf{traj}(C), p)\}$ the sampled trajectory on input variables.*

- *Denote by* $\mathbf{straj_u}(C) = \{\sigma | \sigma \in \mathbf{sampled}_{sr}(\mathbf{traj}(C), p) \wedge \sigma \downarrow V_I = u\}$ *the set of trajectories with sampled input trajectory* u.
- *Denote by* $\mathbf{ssubtraj_u}(C) = \{\sigma | \exists_{\sigma' \in \mathbf{straj_u}(C)} : \sigma \leq \sigma'\}$ *the set of prefixes of the set of trajectories in* $\mathbf{straj_u}(C)$.

Suppose $j = u.ltime$. *Let, for all* $\sigma \in \mathbf{straj_u}(C)$, $\mathcal{TC}_\sigma \in hTests_{sr}(C \textbf{ after } \Sigma_{\sigma'}, sr - \mathbf{mod}(sr + \sigma.ltime - p, sr))$, *and let, for all* $\sigma' \in \mathbf{ssubtraj_u}(C)$ *and* $o \in A_O \cap \mathbf{out}(C \textbf{ after } \Sigma_{\sigma'})$, $\mathcal{TC}_{\sigma' o} \in hTests_{sr}((C \textbf{ after } \Sigma_{\sigma'}) \textbf{ after } o, sr - \mathbf{mod}(sr + \sigma.ltime - p, sr))$, *then*

$$
\begin{array}{l}
\sum\{\sigma; \mathcal{TC}_\sigma | \sigma \in \mathbf{straj_u}(C)\} + \\
\sum\{\sigma; \mathbf{fail} | \sigma \downarrow V_I = u \wedge \sigma \notin \mathbf{ssubtraj_u}(C)\} + \\
\sum\{o; \mathcal{TC}_o | o \in A_O \cap \mathbf{out}(C)\} + \\
\sum\{o; \mathbf{fail} | o \in A_O \backslash \mathbf{out}(C)\} + \\
\sum\{\sigma'; o; \mathcal{TC}_{\sigma' o} | \\
\qquad \sigma' \in \mathbf{ssubtraj_u}(C) \wedge \sigma'.ltime < j \wedge o \in A_O \cap \mathbf{out}(C \textbf{ after } \Sigma_{\sigma'})\} + \\
\sum\{\sigma'; o; \mathbf{fail} | \\
\qquad \sigma' \in \mathbf{ssubtraj_u}(C) \wedge \sigma'.ltime < j \wedge o \in A_O \setminus \mathbf{out}(C \textbf{ after } \Sigma_{\sigma'})\}
\end{array}
$$

is an element of $hTests_{sr}(C, p)$.

The set of tests from a specification $\mathcal{S}$ is defined as $hTests_{sr}(\mathcal{S}, 0)$.

This notion of test allows to apply input actions in between samples and to observe output actions between samples. At the moment in time the actions occur, the sampled continuous variables do not have a real value (since the variables have value $\perp$). Normally this is not a problem except in case an output action is triggered by a specific input sample. For instance, a thermostat might be implemented such that the heater should switch on at exactly 15 degrees Celcius. If the initial temperature of the specified room is 20 degrees Celcius, the (specified) room temperature decreases with 0.3 degrees per minute, and the sample rate is 1 minute, then the implementation will be stimulated with a room temperature of 15.1 degrees Celcius followed by a temperature of 14.8 degrees Celcius. The room temperature that triggers the output action is not applied and therefore not observed. If the specifications correctly models that behavior that at exactly 15 degrees Celcius the output action has to occur, the test leads to the verdict fail because the output action was not observed while it should have been observed. The solution for this problem is to use dynamic sampling which, in case of urgent output actions, generates and sends a sample to the implementation in order to try to trigger the urgent output action.

Definition 6.6. *The* synchronous composition *of a sampled test* $\mathcal{TC} = (T, t_0, A \cup \Sigma_{sr}, \rightarrow \cup \rightsquigarrow)$ *and an implementation* $\mathcal{I} = (S, s_0, A_\tau \cup \Sigma, \rightarrow_{\mathcal{I}} \cup \rightsquigarrow_{\mathcal{S}})$, *denoted by* $\mathcal{TC} \parallel_{sr} \mathcal{I}$ *is defined by*

$$\mathcal{TC} \parallel_{sr} \mathcal{I} = (T \times S, (t_0, s_0), A \cup \Sigma_{sr}, \rightarrow' \cup \rightsquigarrow')$$

with $\rightarrow' =$

$$
\begin{array}{l}
\{((t, s), a, (t', s')) | t \xrightarrow{l} t' \wedge s \xrightarrow{a}_{\mathcal{I}} s' \wedge a \in A \setminus \{\tau\}\} \cup \\
\{((t, s), \tau, (t, s')) | s \xrightarrow{\tau}_{\mathcal{I}} s'\}
\end{array}
$$

and $\rightsquigarrow' =$

$$\{((t,s),\sigma,(t',s'))|\sigma \in \Sigma_{sr} \wedge t \overset{\sigma}{\rightsquigarrow} t' \\ \exists_{p\in\mathbb{R}^{>0},\sigma'\in\Sigma} : 0 < p \leq sr \wedge \mathbf{sampled}_{sr}(\sigma',p) = \sigma\}.$$

Definition 6.7. *Let* $\mathcal{TC} = (T, t_0, A \cup \Sigma_{sr}, \rightarrow \cup \rightsquigarrow)$ *be a sampled test and let* $\mathcal{I} = (S, s_0, A_\tau \cup \Sigma, \rightarrow_{\mathcal{I}} \cup \rightsquigarrow_{\mathcal{S}})$ *be an implementation. The set of test runs, denoted by* $testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$*, is the set of all traces that lead to a state* **pass** *or* **fail***:*

$$testruns(\mathcal{TC} \parallel_{sr} \mathcal{I}) =$$

$$\{\alpha \in traces(\mathcal{TC} \parallel_{sr} \mathcal{I})|\exists_{s\in S} : (t_0,s_0) \overset{\alpha}{\Rightarrow} (\mathbf{pass},s) \ \vee \ (t_0,s_0) \overset{\alpha}{\Rightarrow} (\mathbf{fail},s)\}$$

We say a hybrid implementation passes a hybrid test if only the verdict **pass** is reachable in the execution of the test.

Definition 6.8. *Let* $\mathcal{TC} = (T, t_0, A \cup \Sigma_{sr}, \rightarrow \cup \rightsquigarrow)$ *be a sampled test and let* $\mathcal{I} = (S, s_0, A_\tau \cup \Sigma, \rightarrow_{\mathcal{I}} \cup \rightsquigarrow_{\mathcal{S}})$ *be an implementation, then* $\mathcal{I}$ **passes** $\mathcal{TC}$ *is defined as*

$$\mathcal{I} \textbf{ passes } \mathcal{TC} \iff \forall_{\alpha\in testruns(\mathcal{TC}\parallel_{sr}\mathcal{I})} : \exists_{s\in S} : (t_0,s_0) \overset{\alpha}{\Rightarrow} (\mathbf{pass},s).$$

Let $\mathcal{S}$ *be a specification, then*

$$\mathcal{I} \textbf{ passes } hTests_{sr}(\mathcal{S},0) \iff \forall_{\mathcal{TC}\in hTests_{sr}(\mathcal{S},0)} : \mathcal{I} \textbf{ passes } \mathcal{TC}.$$

6.3.3 Soundness and Exhaustiveness Proofs for Hybrid Tests

The notion of sampled test is sound and exhaustive with respect to the sampled conformance relation.

Theorem 6.9. *(soundness) Let* $\mathcal{I}$ *be an implementation and let* $\mathcal{S}$ *be a specification, then:*

$$\mathcal{I} \textbf{ hioco}_{sr} \ \mathcal{S} \implies \forall_{\mathcal{TC}\in hTests_{sr}(\mathcal{S},0)} : \mathcal{I} \textbf{ passes } \mathcal{TC}.$$

We will sketch a proof for this theorem in the line of the soundness proof for hybrid tests as described in Section 4.6. We assume that there is a test run that leads to the verdict **fail** for an implementation, and we prove that this can only be the case if the implementation does not conform to the specification from which the test is derived. This contradicts the assumption that the implementation conforms to the specification.

In order to prove this we first need to prove that if there is a test run that leads to the verdict **fail** there is a subtest, which by either an output action, a trajectory, or a trajectory followed by an action leads to **fail**. Then, we prove that if a sampled trace leads to the verdict fail, this is either because of an output action that is not allowed

by the specification or by a sampled trajectory that is not allowed by the specification. Because the trace leads to the verdict fail for the implementation we know that this output action or sampled trajectory was allowed by the implementation and therefore the implementation does not conform to the specification.

Sketch of proof. Soundness of sampled conformance is proven by contraposition. Let $\mathcal{S} = (S_{\mathcal{S}}, s_0, A_\tau \cup \Sigma_{sr}, \rightarrow_{\mathcal{S}} \cup \rightsquigarrow_{\mathcal{S}})$ be a specification, and let $\mathcal{I} = (S_{\mathcal{I}}, i_0, A_\tau \cup \Sigma, \rightarrow_{\mathcal{I}} \cup \rightsquigarrow_{\mathcal{I}})$ be an implementation. Let $\mathcal{TC} = (T, t_0, A \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be a test, with $\mathcal{TC} \in hTests_{sr}(\mathcal{S}, 0)$ and suppose that, for some $\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$, $t_0 \stackrel{\alpha}{\Rightarrow}$ **fail**. We show that this α can only exist if $\mathcal{I}$ **hiόco**$_{sr}$ $\mathcal{S}$.

Similar to Lemma 4.25 in Section 4.6 it holds that if $\mathcal{TC} \in hTests_{sr}(\mathcal{S}, 0)$ and $\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$ such that $\mathcal{TC} \stackrel{\alpha}{\Rightarrow}$ **fail**, then there exists a sampled trace α', a subtest $\mathcal{TC}'$, a set of continuous traces $\Omega = \{\omega | \omega \in traces(\mathcal{S}) \wedge \textbf{sampled}_{sr}(\omega, p) = \alpha'\}$ and an output action $o \in A_O$ or a trajectory sampled σ such that $\mathcal{TC}' \in hTests_{sr}(\mathcal{S}$ **after** $\Omega, 0)$ and $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$ and either:

1. **ntrace**$(\alpha) = \alpha' o$ and $\mathcal{TC}' \stackrel{o}{\rightarrow}$ **fail**;
2. **ntrace**$(\alpha) = \alpha' \sigma$ and $\mathcal{TC}' \stackrel{\sigma}{\rightsquigarrow}$ **fail**; or
3. **ntrace**$(\alpha) = \alpha' \sigma o$ and $\mathcal{TC}' \stackrel{\sigma o}{\Rightarrow}$ **fail**.

This can be proven by induction the structure of $\mathcal{TC}$ and α.

For $\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$ and $\mathcal{TC} \stackrel{\alpha}{\Rightarrow}$ **fail** we can then prove the following.

- Suppose **ntrace**$(\alpha) = \alpha' o$, $\Omega = \{\omega | \textbf{sampled}_{sr}(\omega, 0) = \alpha'\}$, $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$, and $\mathcal{TC}' \stackrel{o}{\rightarrow}$ **fail**. Then, because $\mathcal{TC}' \in hTests_{sr}(\mathcal{S}$ **after** $\Omega)$ it follows that $o \notin$ **out**$(\mathcal{S}$ **after** $\Omega)$. Because $\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$ we know that

  $$o \in \textbf{out}(\mathcal{I} \textbf{ after } \Omega).$$

  Therefore, $\mathcal{I}$ does not conform to $\mathcal{S}$.

- Suppose **ntrace**$(\alpha) = \alpha' \sigma o$, $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$ and $\mathcal{TC}'$ is the last subtest (as described in Section 4.6) of $\mathcal{TC}$ and $\mathcal{TC}' \stackrel{\sigma o}{\Rightarrow}$ **fail**, $\Omega = \{\omega | \textbf{sampled}_{sr}(\omega, 0) = \alpha'\}$, and $\Sigma_\sigma = \{\sigma' | \textbf{sampled}_{sr}(\sigma', \textbf{mod}(\alpha'.ltime, sr)) = \sigma\}$. Then, because $\mathcal{TC}' \in hTests_{sr}(\mathcal{S}$ **after** $\Omega)$ and $\mathcal{TC}'$ is described by case for of our notion of sampled test, we know that $o \notin$ **out**$((\mathcal{S}$ **after** $\Omega)$ **after** $\Sigma_\sigma)$. Because $\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$ we know that $o \in$ **out**$((\mathcal{I}$ **after** $\Omega)$ **after** $\Sigma_\sigma)$. Therefore, $\mathcal{I}$ does not conform to $\mathcal{S}$.

- Suppose **ntrace**$(\alpha) = \alpha' \sigma$ and $\Omega = \{\omega | \textbf{sampled}_{sr}(\omega, 0) = \alpha'\}$, $\mathcal{TC} \stackrel{\alpha'}{\Rightarrow} \mathcal{TC}'$, and $\mathcal{TC}' \stackrel{\sigma}{\rightsquigarrow}$ **fail**. Then $\mathcal{TC}'$ is defined by either case 3 or case 4 of our notion of sampled test.

– If $\mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \Omega) = \emptyset$ then $\xi \notin \mathbf{out}(\mathcal{S}\ \mathbf{after}\ \Omega)$. Because

$$\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$$

we know that $\mathbf{traj}(\mathcal{I}\ \mathbf{after}\ \Omega) \neq \emptyset$ and thus $\xi \in \mathbf{out}(\mathcal{I}\ \mathbf{after}\ \Omega)$. Therefore, $\mathcal{I}$ does not conform to $\mathcal{S}$.

– If $\mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \Omega) \neq \emptyset$ then, with

$$\Sigma_\sigma = \{\sigma' | \mathbf{sampled}_{sr}(\sigma', sr - \mathbf{mod}(\alpha'.ltime, sr)) = \sigma'\},$$

$\Sigma_\sigma \cap \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \Omega)) = \emptyset$. Because $\alpha \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$ we know that $\Sigma_\sigma \cap\ \mathbf{sampled}_{sr}($
$\mathbf{traj}(\mathcal{I}\ \mathbf{after}\ \Omega)) \neq \emptyset$. Therefore, $\mathcal{I}$ does not conform to $\mathcal{S}$.

Theorem 6.10. *(exhaustiveness) Let $\mathcal{I}$ be an implementation and let $\mathcal{S}$ be a specification, then:*

$$\mathcal{I}\ \mathbf{hi\not{o}co}_{sr}\ \mathcal{S} \implies \exists_{\mathcal{TC} \in hTests_{sr}(\mathcal{S})} : \mathcal{I}\ \ \mathbf{pa\not{s}ses}\ \ \mathcal{TC}.$$

Sketch of proof. This theorem is proven in line with the proof of exhaustiveness for hybrid conformance in Chapter 4. Namely, suppose that $C_\mathcal{I} \subseteq S_\mathcal{I}$ and $C_\mathcal{S} \subseteq S_\mathcal{S}$ and, for $\alpha \in traces(\mathcal{S})$, $\Omega = \{\alpha' | \alpha' = \mathbf{sampled}_{sr}(\alpha, 0)\}$ and suppose that $0 < p \leq sr$ is the first sample point after trace α. If

$$\mathbf{out}(C_\mathcal{I}\ \mathbf{after}\ \Omega)\ \not\subseteq\ \mathbf{out}(C_\mathcal{S}\ \mathbf{after}\ \Omega) \vee$$

$$\mathbf{infilter}(\mathbf{sampled}_{sr}(\mathbf{traj}(C_\mathcal{I}\ \mathbf{after}\ \Omega), p), \mathbf{sampled}_{sr}(\mathbf{traj}(C_\mathcal{S}\ \mathbf{after}\ \Omega), p))\ \not\subseteq$$

$$\mathbf{sampled}_{sr}(\mathbf{traj}(C_\mathcal{S}\ \mathbf{after}\ \Omega), p).$$

then there exists a test such that the trace $\mathbf{sampled}_{sr}(\alpha\mu, p)$ leads to the verdict **fail**. The theorem follows for $C_\mathcal{I} = \{i_0\}$ and $C_\mathcal{S} = \{s_0\}$ and $p = 0$.

- Suppose that $\alpha = \epsilon$.
  - Suppose that there exists an $o \in \mathbf{out}(C_\mathcal{I}) \setminus \mathbf{out}(C_\mathcal{S})\ \cap\ A_O$.
    - If $\mathbf{traj}(C_\mathcal{S}) = \emptyset$, then a test defined by case 3 of our notion of sampled test leads to the verdict **fail** for output action o.
    - If $\mathbf{traj}(C_\mathcal{S}) \neq \emptyset$, then a test defined by case 4 of our notion of sampled test leads to the verdict **fail** for output action o.
  - Suppose that there exists a σ in

    $$\mathbf{infilter}(\mathbf{sampled}_{sr}(\mathbf{traj}(C_\mathcal{I}), p), \mathbf{sampled}_{sr}(\mathbf{traj}(C_\mathcal{S}), p)) \setminus$$

    $$\mathbf{sampled}_{sr}(\mathbf{traj}(C_\mathcal{S}), p).$$

    Then, with $u = \sigma \downarrow V_I$ and $j = u.ltime$, case 4 of our notion of sampled test leads to the verdict **fail**.

- Suppose that $\alpha = \nu\alpha'$ and suppose there exists a $\mathcal{TC}$ with initial state t_0, and $\mu \in (A_O \cup \Sigma)$ and $\beta = \mathbf{sampled}_{sr}(\alpha'\mu, \mathbf{mod}(\beta.ltime + p, sr))$ and $\beta \in testruns(\mathcal{TC} \parallel_{sr} \mathcal{I})$ and $c, c' \in C_{\mathcal{I}}$ such that $(t_0, c) \stackrel{\beta}{\Rightarrow} (\mathbf{fail}, c')$.
  - Suppose that $\nu = i$ then a test defined by case 2 of our notion of sampled test with $\mathcal{TC}_i = \mathcal{TC}$ leads to the verdict **fail** for trace $i\beta$.
  - Suppose that $\nu = o$ and $\mathbf{traj}(\mathcal{S}) = \emptyset$, then a test defined by case 3 of our notion of sampled test with $\mathcal{TC}_o = \mathcal{TC}$ leads to the verdict **fail** for trace $o\beta$.
  - Suppose that $\nu = \sigma$ then a test defined by case 4 of our notion of sampled test with $u = \sigma \downarrow V_I$ and $j = u.ltime$ and $\mathcal{TC}_\sigma = \mathcal{TC}$ leads to the verdict **fail** for trace $\mathbf{sampled}_{sr}(\sigma, \mathbf{mod}(\sigma.ltime + \beta.ltime + p, sr))\ \beta$.

6.3.4 Comparison with Hybrid Input-Output Conformance

Let $\mathcal{I}$ be an hybrid implementation and let $\mathcal{S}$ be a hybrid specification. Then:

$$\mathcal{I}\ \mathbf{hioco}_{sr}\ \mathcal{S} \text{ does not imply } \mathcal{I}\ \mathbf{hioco}\ \mathcal{S}.$$

We show this by an example.

Example 6.11. *Consider the HTS fragments depicted in Figure 6.4.*

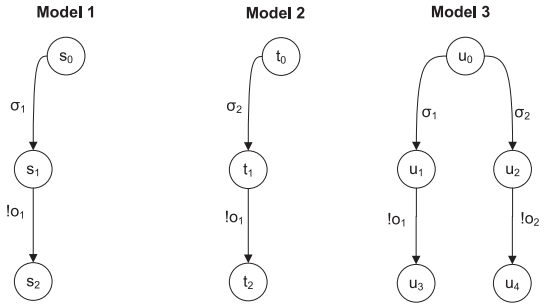

Figure 6.4: HTS Fragments for which Sampling Matters

Let **Model 1** *be a fragment of a specification and let* **Model 2** *be a fragment of an implementation. Suppose* $\sigma_1 \downarrow V_I = \sigma_2 \downarrow V_I$ *and* $\sigma_1 \downarrow V_O \neq \sigma_2 \downarrow V_O$ *while* $\mathbf{sampled}(\sigma_1, p) = \mathbf{sampled}(\sigma_2, p)$ *for some sample point* $0 < p \leq sr$. $\mathcal{I}$ $\mathbf{hioco}_{sr}$ $\mathcal{S}$ *because* $\{\mathbf{sampled}_{sr}(\sigma_2, p)\} \subseteq \{\mathbf{sampled}_{sr}(\sigma_1, p)\}$. *However,* $\mathcal{I}$ $\mathbf{hi\not{o}co}$ $\mathcal{S}$ *because* $\{\sigma_2\} \not\subseteq \{\sigma_1\}$.

By sampling we always lose observations on the implementation, namely valuations. However (obviously), how higher the sample rate, how closer we get. In practice this can be seen as just an extra limitation of testing since it is already impossible to execute all tests that can be derived from a specification. We leave it as

an open problem to determine the conditions under which sampled input-output conformance implies hybrid input-output conformance.

Let $\mathcal{I}$ be an hybrid implementation and let $\mathcal{S}$ be a hybrid specification. Then:

$$\mathcal{I} \textbf{ hioco } \mathcal{S} \text{ does not imply } \mathcal{I} \textbf{ hioco}_{sr} \mathcal{S}.$$

We show this again by an example.

Example 6.12. *Consider the HTS fragments depicted in Figure 6.4. Let* **Model 1** *be a fragment of a specification and let* **Model 3** *be a fragment of an implementation. Suppose* $\sigma_1 \downarrow V_I \neq \sigma_2 \downarrow V_I$ *while* $\textbf{sampled}(\sigma_1, p) = \textbf{sampled}(\sigma_2, p)$ *for some initial sample point* $0 < p \leq sr$*. Then* $\mathcal{I}$ **hioco** $\mathcal{S}$ *because*

$$\textbf{out}(\mathcal{I} \textbf{ after } \sigma_1) \subseteq \textbf{out}(\mathcal{S} \textbf{ after } \sigma_1).$$

However, $\mathcal{I}$ $\textbf{hi}\not{\textbf{o}}\textbf{co}_{sr}$ $\mathcal{S}$ *because* $\textbf{out}(\mathcal{I} \textbf{ after } \{\sigma_1, \sigma_2\}) \not\subseteq \textbf{out}(\mathcal{S} \textbf{ after } \sigma_1)$.

In general, hybrid input-output conformance does not imply sampled input-output conformance because the sampled input can correspond with more than one trajectory in the implementation. However, with certain constraints on the specification hybrid input-output conformance does imply sampled input-output conformance. We formalize these constraints (as weak as possible) after which, we prove this theorem.

We impose two constraints on the specification:

1. whenever a state allows an input trajectory σ then that state allows all input trajectories that are similar to σ modulo sampling with sample rate sr and initial sample point p;
2. all trajectories that are similar modulo sampling with sample rate sr and initial sample point p, lead to the same state.

Definition 6.13. *Let* sr *be a sample rate. Two trajectories* $\sigma, \sigma' \in \Sigma$ *are* similar modulo sampling *with sample rate* sr *and with initial sample point* $0 < p \leq sr$*. denoted by* $\sigma \sim_{sr,p} \sigma'$*, if* $\textbf{sampled}_{sr}(\sigma, p) = \textbf{sampled}_{sr}(\sigma', p)$*. Two traces* α *and* α' *are* similar modulo sampling *with sample rate* sr *and with initial sample point* $0 < p \leq sr$*, denoted by* $\alpha \sim_{sr,p} \alpha'$ *if* $\textbf{sampled}_{sr}(\alpha, p) = \textbf{sampled}_{sr}(\alpha', p)$.

Definition 6.14. *We define an* input trajectory *as a trajectory on input variables only. That is, an input trajectory is a function* $(0, d] \to Val(V_I)$*. Let* $\mathcal{H} = (S, s_0, A_\tau \cup \Sigma, \to \cup \rightsquigarrow)$ *be a HTS. The set of all input trajectories allowed by state* $s \in S$*, denoted by* $IT(s)$ *we define as:*

$$IT(s) = \{\sigma \downarrow V_I | s \overset{\sigma}{\rightsquigarrow}\}$$

For a set of states $C \subseteq S$*, we define:*

$$IT(C) = \bigcup_{s \in C} IT(s)$$

Definition 6.15. *Let σ be an input trajectory. We denote by $[\sigma]_{sr}$ the set of all input trajectories similar to σ modulo sampling with sample rate sr and initial sample point $0 < p \leq sr$:*

$$[\sigma]_{sr} = \{\sigma' | \sigma \sim_{sr,p} \sigma'\}.$$

Definition 6.16. *A state s is* input consistent up to sample rate *sr if, for all $\sigma \in IT(s)$, $[\sigma]_{sr} \subseteq IT(s)$. A HTS $\mathcal{H} = (S, s_0, A \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ is input consistent up to sr if for all $s \in S$, s is input consistent.*

Definition 6.17. *Let sr be a sample rate. A HTS $\mathcal{H} = (S, s_0, A \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ is* trajectory deterministic up to *sr if for all $s, s', s'' \in S$ and $\sigma, \sigma' \in \Sigma$, and for all $0 < p \leq sr$ holds that if $\sigma \sim_{sr,p} \sigma'$ and $s \overset{\sigma}{\rightsquigarrow} s'$ and $s \overset{\sigma'}{\rightsquigarrow} s''$ then $s' = s''$.*

Lemma 6.18. *Let sr be a sample rate. Let $\mathcal{H} = (S, s_0, A \cup \Sigma, \rightarrow \cup \rightsquigarrow)$ be sampled trajectory deterministic up to sr. Then for all $\alpha, \alpha' \in traces(\mathcal{H})$, if $\alpha \sim_{sr,0} \alpha'$ then $\mathcal{H}$ **after** $\alpha = \mathcal{H}$ **after** α'.*

Proof. This lemma is proven by induction on α and α'.

- Suppose $\alpha = \epsilon$ and $\alpha' = \epsilon$ then $\mathcal{H}$ **after** $\alpha = \mathcal{H}$ **after** α'.
- Suppose $a \in A$ and $\alpha = \beta a$ and $\alpha' = \beta' a$ with $\beta \sim_{sr} \beta'$. By the induction hypothesis $\mathcal{H}$ **after** $\beta = \mathcal{H}$ **after** β'. $\mathcal{H}$ **after** $\alpha = \mathcal{H}$ **after** βa means that $\mathcal{H}$ **after** $\alpha = (\mathcal{H}$ **after** $\beta)$ **after** a. Because of the induction hypothesis $\mathcal{H}$ **after** $\alpha = (\mathcal{H}$ **after** $\beta')$ **after** a, which means $\mathcal{H}$ **after** $\alpha = \mathcal{H}$ **after** α'.
- Suppose $\alpha \sim_{sr} \alpha'$ and $\alpha = \beta\sigma$ and $\alpha' = \beta'\sigma'$ and (the induction hypothesis) $\mathcal{H}$ **after** $\beta = \mathcal{H}$ **after** β'. $\mathcal{H}$ **after** $\alpha = \mathcal{H}$ **after** $\beta\sigma$ means that $\mathcal{H}$ **after** $\alpha = (\mathcal{H}$ **after** $\beta)$ **after** σ. Because of the induction hypothesis $\mathcal{H}$ **after** $\alpha = (\mathcal{H}$ **after** $\beta')$ **after** σ. Since $\mathcal{H}$ is sampled trajectory deterministic (and $\sigma \sim_{sr} \sigma$) we know that, for all $s \in S$, if $s \overset{\sigma}{\rightsquigarrow} s'$ then $s \overset{\sigma'}{\rightsquigarrow} s'$. This means, for $C \subseteq S$, that C **after** $\sigma = C$ **after** σ'. Therefore $(\mathcal{H}$ **after** $\beta')$ **after** $\sigma = (\mathcal{H}$ **after** $\beta')$ **after** σ and thus $\mathcal{H}$ **after** $\alpha = \mathcal{H}$ **after** α'.

Theorem 6.19. *Let $\mathcal{I}$ be an hybrid implementation and let $\mathcal{S}$ be a hybrid specification. Let $\mathcal{S}$ be input consistent up to sample rate sr and let $\mathcal{S}$ be sampled trajectory deterministic. Then:*

$$\mathcal{I} \textbf{ hioco } \mathcal{S} \textit{ implies } \mathcal{I} \textbf{ hioco}_{sr} \mathcal{S}.$$

Proof. Suppose that $\mathcal{I}$ **hioco** $\mathcal{S}$. We need to prove that $\mathcal{I}$ **hioco**$_{sr}$ $\mathcal{S}$. That is, for

all $\alpha \in traces(\mathcal{S})$:

$$\mathbf{out}(\mathcal{I} \textbf{ after } \Omega_\alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \Omega_\alpha) \wedge$$

$$\mathbf{infilter}(\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{I} \textbf{ after } \Omega_\alpha), p_\alpha), \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha)) \subseteq$$

$$\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha)$$

with $\Omega_\alpha = \{\omega | \omega \sim_{sr,0} \alpha\}$ and $0 < p_\alpha \leq sr$ is the first sample point after trace α, e.i. $p_\alpha = sr - \mathbf{mod}(\alpha.ltime, sr)$.

1. First we prove that

   $$\mathbf{out}(\mathcal{I} \textbf{ after } \Omega_\alpha) \subseteq \mathbf{out}(\mathcal{S} \textbf{ after } \Omega_\alpha)$$

   Suppose $\mu \in \mathbf{out}(\mathcal{I} \textbf{ after } \Omega_\alpha)$. Note that μ can be either an output action or ξ. Then there is a trace $\alpha' \in \Omega_\alpha$ such that $\alpha \sim_{sr} \alpha'$ and $\mu \in \mathbf{out}(\mathcal{I} \textbf{ after } \alpha')$. Since $\mathcal{I}$ **hioco** $\mathcal{S}$ this means $\mu \in \mathbf{out}(\mathcal{S} \textbf{ after } \alpha')$. It follows that

   $$\mu \in \mathbf{out}(\mathcal{S} \textbf{ after } \Omega_\alpha).$$

2. Second we prove that

   $$\mathbf{infilter}(\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{I} \textbf{ after } \Omega_\alpha), p_\alpha), \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha)$$

   $$\subseteq$$

   $$\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha, p_\alpha))$$

   Suppose $\sigma \in$

   $$\mathbf{infilter}(\mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{I} \textbf{ after } \Omega_\alpha), p_\alpha), \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha).$$

   Then

   (a) $\sigma \in \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{I} \textbf{ after } \Omega_\alpha), p_\alpha)$ and
   (b) there exists a $\sigma' \in \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S} \textbf{ after } \Omega_\alpha), p_\alpha)$ such that $\sigma \downarrow V_I = \sigma' \downarrow V_I$.

   Because $\sigma \in \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{I} \textbf{ after } \Omega_\alpha), p_\alpha)$, there exists a $\sigma^* \in \Sigma$ and an $\alpha^* \in \Omega_\alpha$ such that

   $$\sigma^* \in \mathbf{traj}(\mathcal{I} \textbf{ after } \alpha^*) \text{ and } \mathbf{sampled}_{sr}(\sigma^*, p_\alpha) = \sigma.$$

   We now prove that $\sigma^* \in \mathbf{infilter}(\mathbf{traj}(\mathcal{I} \textbf{ after } \alpha^*), \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha^*))$. In order to prove this we need to prove that there exists a $\sigma^\dagger \in \mathbf{traj}(\mathcal{S} \textbf{ after } \alpha^*)$ such that $\sigma^\dagger \downarrow V_I = \sigma^* \downarrow V_I$. Since S is trajectory deterministic modulo sr, by Lemma 6.18, for all $\alpha' \in \Omega_\alpha$ it holds that $\mathcal{S} \textbf{ after } \alpha' = \mathcal{S} \textbf{ after } \alpha$ and therefore

   $$\mathcal{S} \textbf{ after } \Omega_\alpha = \mathcal{S} \textbf{ after } \alpha^*.$$

So according to (b), there exists a $\sigma' \in \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha^*), p_\alpha)$ such that

$$\sigma \downarrow V_I = \sigma' \downarrow V_I.$$

Then there exists a $\sigma^{**} \in \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha^*)$ such that

$$\mathbf{sampled}(\sigma^{**}, p_\alpha) \downarrow V_I = \sigma' \downarrow V_I.$$

Hence

$$\mathbf{sampled}_{sr}(\sigma^{**}, p_\alpha) \downarrow V_I = traj' \downarrow V_I = \sigma \downarrow V_I = \mathbf{sampled}_{sr}(\sigma^*, p_\alpha) \downarrow V_I.$$

So

$$\sigma^{**} \downarrow V_I \sim_{sr,p_\alpha} \sigma^* \downarrow V_I.$$

Because $\mathcal{S}$ is input consistent it holds that

$$[\sigma^{**} \downarrow V_I] \subseteq IT(\mathcal{S}\ \mathbf{after}\ \alpha^*).$$

Hence, in particular, $\sigma^* \downarrow V_I \in IT(S\ \mathbf{after}\ \alpha^*)$ and this means that there exists a $\sigma^\dagger \in \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha)$ such that $\sigma^\dagger \downarrow V_I = \sigma^* \downarrow V_I$ and thus $\sigma^* \in \mathbf{infilter}(\mathbf{traj}(\mathcal{I}\ \mathbf{after}\ \alpha^*), \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha^*))$. Now, since $\mathcal{I}\ \mathbf{hioco}\ \mathcal{S}$ this means that $\sigma^* \in \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \alpha^*)$. Because $\mathbf{sampled}_{sr}(\sigma^*, p_\alpha) = \sigma$ and $\mathcal{S}\ \mathbf{after}\ \alpha^* = \mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \Omega_\alpha)$ we conclude that $\sigma \in \mathbf{sampled}_{sr}(\mathbf{traj}(\mathcal{S}\ \mathbf{after}\ \Omega_\alpha, p_\alpha))$.

6.4 Implementation Issues and Decisions

Besides the issues described above we also need to resolve four major practical implementation issues. First we need find a finite representation for the sets of allowed input, allowed output, and reachable states of the specification. Otherwise, these sets cannot be computed. Second we need to deal with real time aspects. In practice it takes time to apply, observe and validate the input and output, and there may be delays on the communication medium. Third, we still need to deal with inaccuracies in the application of input (samples) and the observation of output (samples) e.g. caused by rounding off in the adapter. Fourth, we need ways to select input and trajectories and we need to determine when to stop a test. These two issues are related because the same set of criteria can be applied for both. Fifth, we need to connect the test tool to the implementation through the adapter, the medium and the test environment.

6.4.1 Infinite Sets

A hybrid specification has uncountably many transitions, trajectories, and states. The set of trajectories allowed from one state is already uncountable because of trajectory additivity. For a test tool we need finite sets or finite, symbolic, representations of these sets. For instance:

- use differential equations as finite representations of an infinite set of trajectories;
- use $0 \leq \dot{x} \leq 3$ as a finite representation for an infinite set of flow rates for variable x; or
- use the data type of a input message or output message as a finite representation for an infinite set of input actions or output messages.

For our prototype tool we make use of the χ language and hybrid χ simulation tool-set. The χ language allows us to specify differential equations on continuous variables. Together with an initial valuation on the variables and a (possibly infinite) duration of the continuous behavior (after which e.g. a discrete-event takes place), this is finite representation of an infinite set of trajectories; namely the set of all trajectories up to the duration. The χ language also allows to send and receive real values via channels. The variable which is assigned the value that is received, together with the type of the variable form a finite representation of a (mathematically) infinite set; namely the set of real values.

6.4.2 Real Time

With a well chosen sample rate and a fast enough platform on which the test tool is executed, it is possible to generate and execute a test in real time. A well chosen sample rate in this case means a sample rate that leaves sufficient time to execute these operations.

Clock skew is the phenomenon that the system clock of the implementation side runs slightly faster or slower than the system clock of the execution platform of the test tool. An issue is that the samples and actions are applied and observed with a delay both because of the communication and clock skew between the specification side and the implementation side of the test setup in case the implementation and test tool are not executed on the same platform. If the implementation and test tool are executed on the same platform the issue of clock skew because both test tool and implementation use the same system clock and the delays over the medium is reduced compared to using a separate platform for the test tool.

In case the tool and the implementation are not on the same platform, a solution is to add time stamps to the input and output actions and samples. The time stamp attached to the input allows the test environment on the implementation side to handle the application of the input at the time indicated by the time stamp. The time stamp attached to the output indicates the actual time the output was performed by the implementation according to the system clock of the implementation platform. The test tool validates whether the output was allowed at the time indicated by the time stamp.

Unfortunately, there is a problem with this solution. It can be the case that after the test tool sends a scheduled input to the test environment, the test tool observes an output that was actually performed *before* the input is performed. Because of this

output, the already scheduled input is not applied in accordance with the test. This can even occur with relatively large sample rates, because it can be for instance that an output action has to be followed soon after by an input sample. The problem can be reduced, but it cannot be solved, by minimizing the possible delay over the network.

Krichen and Tripakis [31] demand that communication delays and skewed clocks are added to the specification. This results in a more lenient specification, or viewed differently: the causes of the inaccuracy are considered part of the behavior of the implementation. This solution requires explicit knowledge about the maximum delays and measures of inaccuracy (on which the modeler could be wrong) and weakens the strength of the test verdict. Bohnenkamp [10] proposes to embed timing inaccuracy in the conformance relation and notion of test. However, he does not describe how this can be done. We discuss this possibility for hybrid testing in the next section.

6.4.3 Inaccuracy in the Observations

Besides the inaccuracy in testing caused by time delays there are other causes for inaccuracy. We already have discussed that we cannot accurately test with physical behavior. And, with sampling, there is inaccuracy in the observations caused by rounding. All causes for inaccuracy, both when using samples and if we want to use physical behavior itself, require a more lenient specification, or a new conformance relation that takes these inaccuracies into account.

In the hybrid case, we see three kinds of inaccuracy (depicted in Figure 6.5).

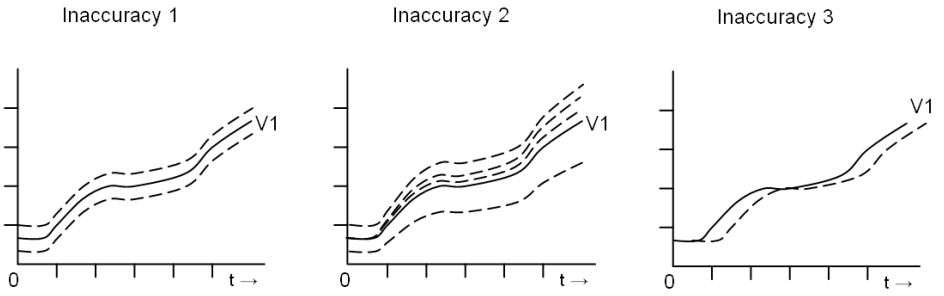

Figure 6.5: Inaccuracy

The first kind of inaccuracy is not cumulative. The bounds of inaccuracy always stay the same and there is no inaccuracy in time. This is the kind of inaccuracy caused by rounding of the observed behavior or caused by limited accuracy of the sensors that observe the continuous behavior. The second kind of inaccuracy is cumulative. When a certain small amount of inaccuracy is observed, this inaccuracy progresses

to the rest of the observations. The third kind of inaccuracy is also cumulative. However, it has the additional problem that it effects the duration of the trajectory. This is the kind of inaccuracy that occurs because of clock drift or delays in the communication over the communication medium.

The problem with a formal conformance relation that takes inaccuracy into account is that inaccuracies influence the behavior of an implementation more radically than sampling. E.g. with sampling we needed to take into account that more trajectories may be characterized by one particular trajectory. Still it can be determined which trajectories these are and it can be determined output behavior has to be considered correct behavior of the implementation. If we take into account inaccuracy we cannot determine anymore how the specification is influenced. Inaccuracy may have the effect that an output action occurs earlier or later than specified (see Figure 6.6).

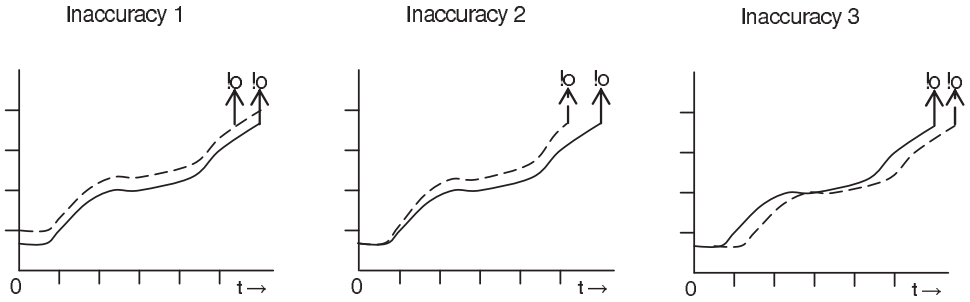

Figure 6.6: Inaccuracy with Output

If the output action was triggered by the value of the continuous input, e.g. if a thermostat turns a heater on at 15 degrees, then this output occurs earlier if due to inaccuracy in the observations the flow rate of the input is higher than specified.

However, an output action may also be triggered by time (e.g. if a thermostat gives a warning after 5 hours of heating a room). In this case, the inaccuracy in the continuous behavior should not influence when the output action occurs. A HTS does not contain the information to distinguish whether the output was triggered by the flow or by time. This makes it impossible to include inaccuracy in a similar way as sampling in our theory.

The solution may be to lift the implementation and specification to a higher level formalism like hybrid automata with guards on the transitions that tell why an output action took place. This remains an interesting challenge for future research. For now the only solution is to demand that the specification is lenient enough to take into account the possible inaccuracies that are caused by the communication between the test engine and the implementation.

6.4.4 Test Selection and Stop Criteria

The test generation algorithm does not prescribe how to select input and when to terminate a test with the verdict pass. These issues are left to the test tool implementation.

In TorX the input can be selected either manually (by the user) or automatically by the test tool.In manual input selection mode the user can select which input action to apply to the implementation. For real time hybrid testing, manual test selection is not a viable option. Obviously, if we test with continuous behavior in real time, there is no time to select new input manually. Even if the test tool uses sampled continuous behavior manual test selection is not a viable option since the sample rate would need to be high enough for the test engineer to select input in between two samples.

Input can be selected automatically, as prescribed by the test generation algorithm. However, in selecting we need to take into account that always eventually (sampled) continuous behavior needs to be selected in order to be able to progress in time. It is possible to select input at random. The selection domain can be big, even infinite, and generate many tests that are not of interest to the user. This domain can be narrowed by specifying a set of values from which the test tool selects the input and a distribution of probabilities that an input occurs. It is also possible to implement test selection according to a test purpose [53]. A test purpose is sequence of actions accordance to which the test is guided. E.g. If the test purpose states that repeatedly after an ”ON” signal always an ”OFF” signal is applied to the implementation, then during test generation input is selected according to this purpose.

Testing can be terminated either manually or automatically. Manually means that the test engineer that uses the tool is in control to stop the test and automatically means that the tests stops when some criterion is met or when some test purpose is fulfilled. E.g. a stop criterion can be a specified duration for the test or a specified amount of coverage of the specification.

6.4.5 Connecting to the Implementation

The connection between the test tool and the implementation consists of three parts: the software adapter on the test tool side, and the communication medium and the test environment on the implementation side. The connection between the test tool and the implementation needs to be customized for the implementation under test.

In general, the way in which input and output is modeled by the specification differs from the actual input and output of the implementation. E.g. an analog electronic signal of the implementation may have been specified as a trigonometric function. The adapter transforms the specified input to a format suitable for the implementation and it transforms output from the implementation to the format used in the specification. It also deals with converting variable names, channel names, and data to interfaces specific to the implementation and the other way around. E.g. an action specified as an on signal or off signal may be implemented

as a boolean value in the implementation. The adapter implements this conversion. The adapter also implements the communication protocols for the medium.

Recall that by the medium we understand all electronics and cables connecting the test tool with the implementation. This includes (specific) hardware that might be needed to transform actions or continuous behavior to electronic signals. Which medium to choose and how to implement it depends highly on the implementation under test.

Sometimes it is desirable that the implementation side of the test setup also needs a customized interface with the medium. The test environment forms the interface between the medium and the implementation. The test environment can also take care of time stamps. It can also implement a filter to test the implementation on part of the output behavior. For instance, if we are interested in testing a thermostat on whether it correctly switches on or switches off a heating device, then we can filter the output of the implementation and only send these outputs to the test tool and not send the (output) messages meant for the user display.

6.5 Prototype Tool

We have implemented a prototype test tool with sampled continuous behavior [41]. This prototype is implemented in Python [42] for the Linux operating system. We implemented our prototype test tool based on the TorX architecture and using the hybrid χ simulation tool set.

Regarding the issues discussed earlier in this chapter we made the following choices.

- We have implemented the on-thy-fly test generation algorithm.
- We have implemented a static sampling strategy. The sample rate can be easily adapted and dynamic sampling strategies can still be implemented.
- A set of trajectories up to a specific duration is represented symbolically by ordinary differential equations with a maximum duration.
- In TorX, the input is restricted by the type of the receive action. Input is selected manually or automatically from the elements of e.g. a message type in PROMELA. We have implemented the same way of input restriction and selection in our prototype tool. The set of input actions of the specification is restricted by the type of the receive channel. Because this set is possibly infinite (e.g. the set of all strings), additionally a list of allowed input actions, (e.g. {on,off}) can be included as a parameter of the test-engine. Then, the input messages are selected from this list.
- The test tool implementation can handle time stamps for actions and samples.
- It is assumed that the specification takes into account inaccuracies e.g. caused by the rounding of values.

- Input actions can be selected manually (for non real-time tests) or automatically, either at random from a subset of input actions, or a simple test purpose can be implemented.
- A test can be terminated at any time by the user of the tool or by specifying a maximum duration of the test.
- An adapter has been made that allows client-server communication over a TCP/IP network using sockets.

6.6 Testing Two Toy Examples

As initial experiments with the prototype test tool we have performed two toy example case studies. In the first case study we tested a water-tank controller simulator and three mutants of that controller against a hybrid specification of the water-tank controller. In Chapter 7 we test a similar (but more complex) industrial controller against a hybrid χ specification. In the second case study we tested a moving robot arm simulator and three mutants of that robot arm against a hybrid specification of the robot arm.

The χ language was chosen as the implementation language of the toy examples as well because the χ tool set provides the possibility to perform distributed simulation of χ processes, possibly together with other pieces of software over a software bus. This made it possible to also connect our test tool to the distributed χ simulator. Because the distributed χ simulator only works for discrete-event χ models, the water-tank controller and robot implementations had to be modeled as discrete-event systems which accepted sampled continuous input behavior and display sampled continuous output behavior.

Because these case studies were about testing the test tool, our approach was that we first made the hybrid χ specifications and based on the specifications we made the implementations. From the implementations, the mutants were created by making small changes to the models.

6.6.1 The Water-tank Controller

The toy example water-tank controller is a controller that observes the temperature behavior of water inside a water tank and turns on or turns off a heater to control the water temperature (see Figure 6.7). This example is a prototypical example of a (software) controller that can be validated with hybrid model-based testing. The temperature flow is continuous input for the controller and the on and off actions are discrete-event output. If the temperature drops below 2.0 ^oC, then the heater is turned on, after which, if the temperature exceeds 10.0 ^oC, the heater is turned off.

The hybrid χ specification of the water-tank controller is shown in Figure 6.9. The process *Controller* specifies the behavior of the controller. The controller receives

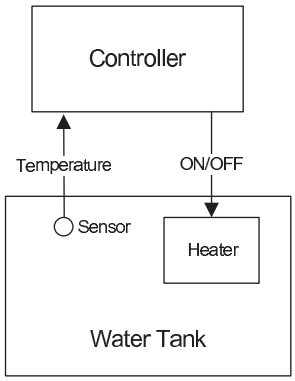

Figure 6.7: Water Tank and Controller

continuous input through the variable T. It represents temperature input that an embedded system controller would receive through a sensor. The action $h!!\text{“}ON\text{”}$ specifies an urgent output action of value ON over channel h. This specifies a message or signal that turns on or turns off a heater. The controller repeatedly waits until the water temperature drops below 2 ^oC and then turns the heater on, and waits until the temperature exceeds 10 ^oC before it turns the heater off.

$$
\begin{array}{l}
\mathsf{proc}\ \mathit{Controller}(\ \mathsf{cont}\ T:\ \mathsf{real},\ \mathsf{chan}\ !h:\ \mathsf{string}\ )\ = \\
[\![\\
::\ *(\ T \leq 2.0 \rightarrow h!!\text{“}ON\text{”}\ [\!]\ T \geq 10.0 \rightarrow h!!\text{“}OFF\text{”}\ ) \\
]\!]
\end{array}
$$

Figure 6.8: Specification for the Water-tank Controller

For simulation, as described in 3, we normally specify the behavior of the Water tank as another process and compose these two processes into a model using the parallel composition. For testing, we test an implementation of the controller against the specification of the controller, without the water tank. However, we still want to specify the continuous input for the test, namely the temperature behavior specified by variable T. Therefore, we compose the process *Controller* together with our desired input behavior into one model.

The temperature input for the controller is modeled by the ordinary linear differential equation $\dot{T} = 3.0 * n - 1.0$. Initially, the discrete variable n has value 0.0. This means that initially $\dot{T} = -1.0$. If $T = 2.0$, then the value 1.0 is assigned to n and therefore $\dot{T} = 2.0$. If $T = 2.0$ then $\dot{T} = -1.0$ again. Note that define channel h as a parameter of the model in order to make the output actions $h!!\text{“}ON\text{”}$ and $h!!\text{“}OFF\text{”}$ observable actions for the prototype test tool.

Next, we made a discrete-event water-tank controller so that we can use the distributed χ simulator. For the experiment, we model this controller such that it waits for input samples instead of requiring input samples according to some sample rate.

```
model TC( chan !h : string ) =
[[ cont T : real = 10.0, var n : real = 0.0
// input restriction
:: Ṫ = 3.0 * n − 1.0
[] *( T ≤ 2 → n := 1.0; T ≥ 10 → n := 0.0 )
// controller specification
[] Controller(T, h)
]]
```

Figure 6.9: Model Composition for the Water-tank Controller

The controller waits for samples of continuous input, which can be given at any sample rate. This allows us to test the controller simulation non real-time, which allows to observe better what happens during the tests. This process is shown in Figure 6.10.

```
proc Controller( chan ?t : real, !h : string ) =
[[ var w : real = 0.0
:: *( t?w; ( w ≤ 2.0 → h!!"ON"
           [] w > 2.0 ∧ w < 10.0 → skip
           [] w ≥ 10.0 → h!!"OFF"
           )
   )
]]
```

Figure 6.10: Implementation for the Water-tank Controller

This water-tank controller repeatedly waits to receive a temperature input, after which it immediately evaluates whether the temperature it received is below 2 ^oC or above 10 ^oC. If the temperature is below 2 ^oC , then it sends an on message as output. if the temperature is above 10 ^oC, then it sends an off message as output. After that, the implementation waits for new temperature input again.

This controller behaves differently then the hybrid χ model. Every time this controller receives a temperature below 2.0 ^oC it produces the output action ON. The hybrid controller is modeled such that after it receives a temperature below 2.0 ^oC it produces the output action ON after which it only waits for the temperature to increases to 10.0 ^oC. However, for the input behavior specified in the hybrid χ model, the output behavior of both models is the same. After producing the action ON when 2.0 ^oC is reached, the temperature always increases to 10.0 ^oC again. Additionally, after producing the action OFF when 10.0 ^oC is reached, the temperature always decreases to 2.0 ^oC again.

We created three mutants of this water-tank controller. These mutants are:

mutant 1: the output action $h!!"ON"$ was changed to $h!!"OFF"$,

mutant 2: the guard $w \leq 2.0$ was changed to $w \leq 3.0$, and

mutant 3: $w \leq 2.0$ was changed to $w \leq 1.0$.

For testing these water-tank controllers, an adapter needed to be implemented that established the connection between the test tool and the software bus of the distributed χ simulator. This adapter implemented a socket connection with the software bus used for the distributed χ model simulator. It also ensures that the samples of input from variable T are correctly received by the implementation via channel t.

From the hybrid specification, the prototype test tool generates tests with sampled continuous behavior and constant sample rate of 1.0 second. The prototype test tool generates and executes tests for the water tank controller according to the on-the-fly test algorithm described in Section 6.2.

Example 6.20. *For the correct water-tank controller implementation, the following test is generated.*

step 1: *Select a temperature decrease of* $-1.0\ ^oC/sec$ *and apply this trajectory to the implementation for 8 seconds (after which the temperature* T *equals* $2.0\ ^oC$*). As long as no output action is observed, the test continues. If an output action is observed stop testing with verdict* **fail**.

step 2 *Observe an output action from the implementation: if* $h!!\text{“}ON\text{”}$ *or no output is observed, then continue; if* $h!!\text{“}OFF\text{”}$ *is observed then conclude verdict* **fail**.

For the correct water-tank controller implementation, the output action $h!!\text{“}ON\text{”}$ *is observed and testing continues.*

step 3: *Select a temperature increase of* $2.0\ ^oC/sec$ *and apply this trajectory to the implementation for 4 seconds (after which the temperature* T *equals* $10.0\ ^oC$*).*

step 4: *Stop testing with the verdict* **pass**.

Testing the correct water-tank controller simulator did not lead to finding any mistake. Because the test captured all possible behavior, this shows that with this particular input our test tool does not return a false negative test verdict. In mutant 1 an incorrect output action was observed. The test tool received the output ON while the output OFF was specified. In mutant 2 an unexpected output action was observed. The test tool received the output ON when variable TS had value 3 and no output action was expected yet. In mutant 3 an output action was expected but it was not observed. The specification expected an output when T had the value 2 but no output was observed.

6.6.2 The Robot Arm

The second example is a robot arm moving on a track (see Figure 6.11). This is a prototypical example of an embedded system device, e.g. a robot arm in an assembly line, or a stage of a waferstepper machine.

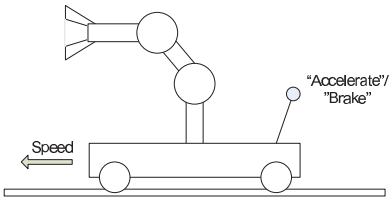

Figure 6.11: A Robot Arm moving on a Track

The robot arm receives the message Accelerate or the message Brake as discrete-event input, e.g. by remote control, and it moves on the track with some speed, which is the continuous output. Depending on which message it receives it either accelerates to its maximum speed and keeps moving at that speed, or it brakes to speed 0.

The hybrid χ specification of the robot arm is shown in Figure 6.12.

```
proc Robot( cont S : real, chan ?c : string ) =
|[ cont T : real = 0.0, var n : real = 0.0, m : string = "STOP"
:: Ṡ = −n ∗ 0.2 ∗ (T − 10.0) || Ṫ = 1.0
|| ∗( c?m; T := 0.0
   ; ( m = "ACC" → n := 1.0; T ≥ 10.0 → n := 0.0
     [] m = "BRK" → n := −1.0; T ≤ 0.0 → n := 0.0
     [] m ≠ "ACC" ∧ m ≠ "BRK" → skip
     )
   )
]|

model R(chan c : real) =
|[ cont S : real = 0.0
:: Robot(S, c)
]|
```

Figure 6.12: Specification of the Robot Arm

The process *Robot* models a robot that moves according to a specified speed. Initially its speed is 0.0. If the robot receives an accelerate message ACC, then the robot starts accelerating. The speed is modeled by the variable S. An additional (clock) variable T is needed to model the acceleration curve. This variable is reset to 0.0 every time the robot starts accelerating or braking. Initially, after the message ACC is received, the robot arm starts accelerating with $\dot{S} = -1.0 * 0.2 * -10 = 2.0\ m/sec^2$. After ten seconds of acceleration, the acceleration is explicitly set to 0.0 otherwise in this model the speed would decrease. Only after that, a new input message may be received. The acceleration curve or braking curve cannot be interrupted by a new message. Furthermore, if e.g. this robot receives an input action BRK but its speed is already 0.0 then its speed will remain 0.0.

Again, we create a discrete-event χ process for the distributed χ simulator to serve as (correct) implementation for our case study. Because the goal of this case study is to test the test tool we wanted to control the input applied to the implementation, and not have it selected for instance at random. Therefore, we use variable T as continuous input for the robot arm. This allows us, like for the water tank, to test the robot arm non real-time. The value of this variable is received over channel c but does not influence the output behavior. The speed of the robot arm is now modeled by variable p and, after a sample of input is received, it is sent over channel s, to the test tool. If an input message is received over channel c, then the robot arm starts either accelerating or braking.

```
proc Robot( chan !s, ?t : real, ?c : string ) =
[[ var n : real = 0.0, i : real = 0.0, u : real = 0.0, m : string = "STOP"
:: *( t?i; p := q + (-n * 0.2 * (u - 10.0)); s!!p ; u := u + 1.0 )
|| *( c?m; u := 0.0
     ; ( m = "ACC" -> n := 1.0; u >= 10.0 -> n := 0.0
       [] m = "BRK" -> n := -1.0; u <= 0.0 -> n := 0.0
       [] m /= "ACC" /\ m /= "BRK" -> skip
       )
     )
]]
```

Figure 6.13: Discrete-event χ Specification of the Robot Arm

For the robot we created three mutants as well. These mutants were:

mutant 1: the assignment to variable p was changed,

mutant 2: the discrete variable n was initialized with value 1.0, and

mutant 3: in the alternative choice statement where the message BRK is received as input action but $u \geq 10$, the variable n was set to 1.0.

The adapter still establishes and maintains the connection to the software bus used by the distributed χ simulator. It also maps the continuous input from variable T and the messages of channel c to the channels t and c of the implementation respectively. The adapter also ensures that the values sent via channel s of the implementation are validated against samples of variable S of the specification, that samples of variable T of the specification are received via channel t by the implementation, and that input actions specified by channel c from the specification are received by the implementation.

From the hybrid specification, the prototype test tool generates tests with sampled continuous behavior and constant sample rate of 1.0 second. The prototype test tool generates and executes tests for the robot arm according to the on-the-fly test algorithm described in Section 6.2.

Example 6.21. *For the correct robot arm, the following test is generated.*

step 1: *The input action c?m is selected. Manually, we select the message ACC. This message is applied to the implementation.*

step 2: *For the duration of* 10 *seconds, the speed of the robot arm implementation is observed. Every second a sample of output is observed and validated against the specified speed: if the observed speed corresponds to the specified speed increase, at the sample points, then the test continues; if the observed speed does not correspond to the specified speed increase at the sample points, then the test terminates with verdict* **fail**.

For the correct implementation, the observed output corresponds to the specified speed. Therefore, testing continues.

step 3: *The input action c?m is selected. Manually, we select the message BRK. This message is applied to the implementation.*

step 4: *For the duration of* 10 *seconds, the speed of the robot arm implementation is observed. Every second a sample of output is observed and validated against the specified speed: if the observed speed corresponds to the specified speed decrease, at the sample points, then the test continues; if the observed speed does not correspond to the specified speed decrease at the sample points, then the test terminates with verdict* **fail**.

For the correct implementation, the observed output corresponds to the specified speed. We terminate the Terminate the test with verdict **pass**.

For the correct implementation these tests did not lead to finding any mistake. This showed that also in this case our test tool does not return a false negative test verdict. Testing the mutants led to finding the following mistakes. In mutant 1 of the robot acceleration went faster than expected. The test tool observed value 5.5 for variable S when it expected the value 3.6 for variable S after 2 seconds. In mutant 2 the robot accelerated unexpectedly. The test tool observed value 1.9 for variable S after two time units when it expected the value 0 for variable S. In mutant 3 the robot accelerated when braking was expected. The test tool observed that after first accelerating fully and applying an input action BRK, that the robot arm accelerated again.

6.7 Concluding Remarks

In this chapter we discussed the possibilities for implementing a hybrid model-based test tool. An on-the-fly test generation and execution algorithm was presented to implement (hybrid) model-based test generation and execution.

The use of samples was also formalized in a new conformance relation and notion of test. For clarity of the definition of a sampled trajectory we only defined sampling with a fixed sample rate. To define complex sampling strategies, only the definition

of a sampled trajectory needs to be adapted to define sampled trajectories according to the strategy.

In practice we have to deal with computation time, delays in communication between test tool and implementation, inaccuracy in measurements and rounding of values. Currently, we cannot embed these causes of inaccuracy into a formal conformance relation and notion of test. At present, the only solution is to make a more lenient specification that allows inaccuracy

A prototype version of a hybrid test tool has been implemented. We have implemented the test generation and execution algorithm as described in Section 6.2 and use sampled behavior as described in Section 6.3. We have run experiments with our prototype tool on two toy example case studies. The next chapter describes how we have used this prototype tool in an industrial setting.

7

Industrial case: The Vacuum System

This chapter describes an industrial case study that we performed with our prototype tool. The goal of this case study was to show that hybrid model based testing can be used in practice. We tested the controller of an exemplary vacuum system of a waferstepper machine.

First we introduce the vacuum system. Then we describe a hybrid χ model of the vacuum system. The hybrid χ model itself is found in Appendix A. This χ model is based on the hybrid χ model that was made by Frank Stappers [45] for a separate case study on a vacuum system. Finally we describe the results of testing the implementation of the vacuum controller against a simplified hybrid χ specification.

7.1 The Vacuum System

A waferstepper is a complex lithography machine that etches images of chip layouts on silicon wafers with nanometer precision. The newest generation of these machines uses Extreme Ultra Violet (EUV) laser light for printing the image on the wafers. Because gasses can absorb the light before it reaches the wafer and gasses can cause damage to the fragile components of the machine, the lithography process has to take place in vacuum.

An example vacuum control system of an EUV machine (depicted in Figure 7.1) supervises multiple vacuum chambers. These chambers contain pumps and valves to change the vacuum conditions. Pumps are used to remove air from a chamber and thus create vacuum. Valves are used to let air into the chamber and thus create atmospheric conditions. The pressure conditions in each chamber are controlled by a software component that is implemented in Labview and runs on an ordinary Windows PC. The main task of this controller is to turn on or turn off pumps, and to close or open valves. A system controller executed on a SUN workstation supervises all the subsystem controllers. This SUN machine is for instance used to start pumping down a chamber or venting a chamber.

Figure 7.2 depicts the hardware of one exemplary vacuum subsystem. If the chamber is pumped down to vacuum conditions, then first the normal pump is turned on and

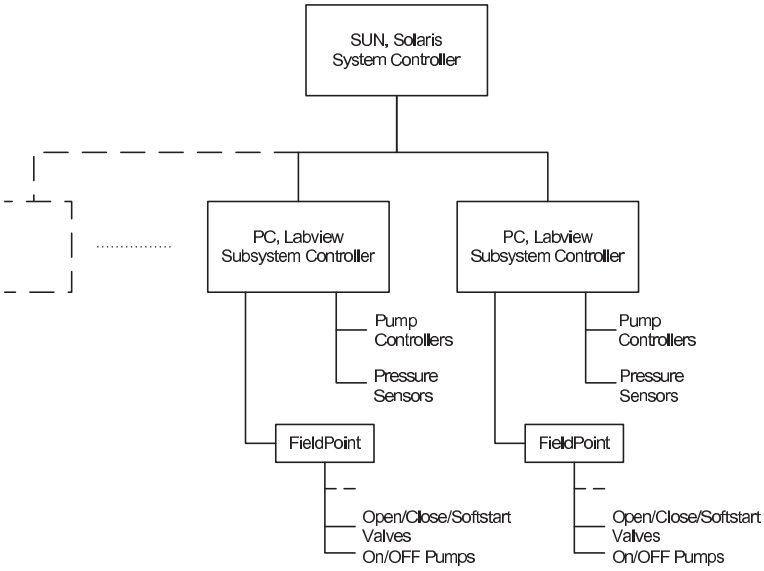

Figure 7.1: Vacuum Control Architecture

the valve is half way opened. This position of the valve is called softstart. If the pressure in the chamber drops below a specific level and the normal pump cannot decrease the pressure further, then the turbo pump is turned on and the valve is fully opened. If the chamber is vented from vacuum conditions, then first the pumps are turned off and the valve is closed. After that, the air valve is opened and air flows into the chamber from the pressurized air container. The chamber is never fully closed off. There are leaks. This means that if the pressure inside the chamber is lower than the pressure outside the chamber then some air flows into the chamber. If the pressure outside the chamber is lower than the pressure inside the chamber then some air flows out of the chamber.

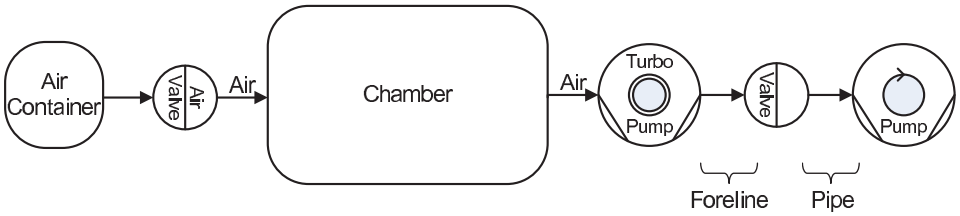

Figure 7.2: Vacuum Hardware

The pressure inside the chamber, inside the pipe and inside the foreline are determined by the state of the pumps, valves, and the size of the leaks. E.g. if the

pressure in the air container is higher than the pressure in the chamber and the air valve is opened, then air flows from the container into the chamber to form an equilibrium. If the pressure in the pipe is lower than the pressure in the foreline and the valve is open but the pumps are off, then air flows from the foreline to the pipe and from the chamber to the foreline until an equilibrium is reached in all three compartments.

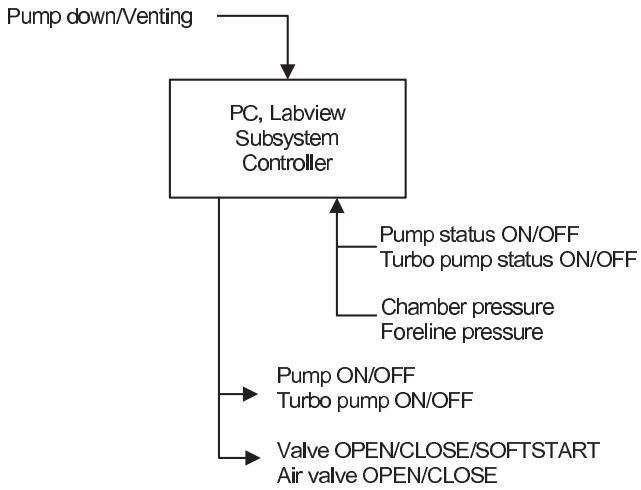

Figure 7.3: Subsystem Controller Input and Output

The controller observes the pressure in a chamber through multiple sensors and reads the status of the pumps. Each sensor is capable of measuring the continuous pressure within a specific range. The controller stores pressure samples with a sample rate of one second. Thus, it is possible to test the controller by using sampled continuous behavior.

The subsystem controller, upon request by the system controller, pumps down or vents the chamber by performing a sequence of actions (see Figure 7.4 and 7.5).

Note that the venting sequence contains a loop (between step 5 to step 11 of the sequence). The chamber pressure after venting is kept a little bit above atmospheric levels. Thus, after venting air flows out from the chamber through leaks. If a lower threshold is reached (step 10) the air valve is opened again to let air into the chamber until the upper threshold is reached, after which the air valve is closed and the pressure decreases again until the lower threshold is reached.

pump down sequence

```
 1. Close air valve
 2. switch on primary pump
    Wait until primary pump on or pump time out
    Softstart foreline valve
 3. Wait until chamber pressure < softstart limit
 4. Open foreline valve
 5. Wait until chamber pressure < turbo pump limit
 6. Switch on turbo pump
```

Figure 7.4: Pump Down Sequence

venting sequence

```
 1. Switch off turbo pump
 2. Close foreline valve
 3. Turn off primary pump
 4. Wait until turbo pump stopped
 5. Open air valve
 6. Wait until chamber pressure > foreline pressure
 7. Open foreline valve
 8. Wait until chamber pressure > final atmosphere +
    upper treshhold
 9. Close air valve
    Close foreline valve
10. Wait until chamber pressure < final atmosphere +
    lower treshhold
11. Goto 5
```

Figure 7.5: Venting Sequence

7.2 Some Choices

Modeling the complete vacuum system (containing the system controller, all the subsystem controllers, and all the chambers) is too complex and defies the purpose of this case study. Therefore, only the subsystem controller of one chamber is tested. Furthermore, because our prototype is not optimized for computation speed and needs sufficient time to compute the sets of transitions allowed and validate the output of implementation, we concentrate on the basic functional behavior of a subsystem controller. Namely, we plan to test whether the Labview controller correctly performs the sequences of actions required to pump down the chamber to vacuum conditions and the sequence of actions required to vent the chamber to atmospheric pressure. Other behavior, like error handling, is not considered. For instance, error messages are not observed from the controller. Furthermore, the messages to the pumps and valves are sent every second. To reduce the number of output messages

to be validated we decided to test only for change in output messages. That is, we implemented a test environment such that it only communicates changes in these messages to the test tool. For instance, if a message to turn off the pump is followed by a message to turn on the pump, then only the message to turn the pump on is communicated. If a message to turn off the pump is followed by another message to turn off the pump then the (second) message to turn off the pump is not communicated.

7.3 Specification of a Vacuum Subsystem in Hybrid χ

In this section we describe the χ specification of the vacuum system. The complete specification can be found in Appendix A. This specification was eventually not used for testing. The reason was that at the time this case study was performed, the χ tool set was still under development as well and could not handle guarded differential equations as used in this model. This model is presented here to show how such system should be modeled in hybrid χ. First we describe the structure of the hybrid χ model. Then, we describe each process of the hybrid χ model in more detail. As it turned out, this specification was too complex to be used in practice. However, we still describe it here as a reference specification for the vacuum control system and testing.

We divide the specification into a controller part and the environment of the controller. The controller part specifies the output behavior of the subsystem controller. The environment part specifies the input behavior of the subsystem controller. Note that both parts are contained in one specification. The controller is modeled by one process. The environment consists of a system controller process, pump and valve processes, and the pressure flows. Sensors do not need to be modeled as a separate process. This is because we do not want to make a difference between the pressure in the chamber and the pressure observed by the controller. However, e.g. if we would like to specify that switching between sensors of a different pressure range should cause a disturbance in correctly observing the pressure, then sensors would be specified additionally.

Figure 7.6 depicts the processes of the χ specification. The pumps and valves influence the pressure in the foreline, pipeline and chamber. Inside each of these processes we specify the influence of the component on the pressure behavior. E.g. Cp_{av} models the influence of the airvalve on the chamber pressure Cp. The chamber pressure, foreline pressure, and pipeline pressure are then specified by the sum of pressure influences of each component. The leakage of air out of the chamber or the leakage of air into the chamber is modeled by another pressure variable Lkp. This leakage influences the chamber pressure as well.

The subsystem controller receives pump status messages (being either on or off) from the pumps and sequence start messages (being either pumpdown or venting) from the system controller. These are modeled by receive actions and depicted by ingoing arrows in Figure 7.6.

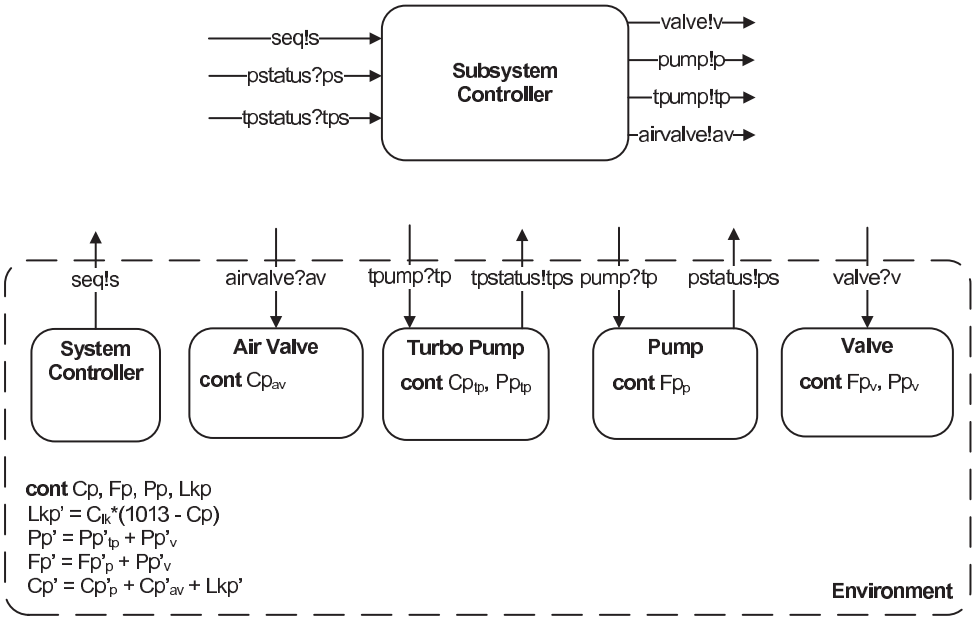

Figure 7.6: Processes of a Vacuum System Specification

The subsystem controller sends message on or the message off to the pumps and message open, softstart or close to the valves. These are modeled by send actions and depicted by outgoing arrows in Figure 7.6. The system controller sends sequence start messages to the subsystem controller. The pumps receive message on or off and send status messages, being on or off, back to the controller. The valves only receive message open, close, or softstart from the subsystem controller.

7.3.1 The Environment

The environment consists of five processes (S, Valve, AirValve, Pump and TurboPump). Below we describe briefly how each process is specified. The complete specification is given in Appendix A.

S: This process describes the relevant behavior of the SUN system controller. For our case study only selecting the start of a venting sequence or selecting the start of a pump down sequence is relevant. The process S in Appendix A repeatedly switches between starting the pump-down sequence and starting the venting sequence. Because we specify this by non-urgent channel communications, this process specifies that a new sequence can be started at any time. It is also possible to define a system controller that starts a sequence in a different way. For instance, it is possible to specify a system controller that waits a fixed amount of time before starting a sequence.

Air Valve: This process specifies the valve that lets air into the chamber if the chamber is vented. The process $AirValve$ in Appendix A specifies a valve which is either fully opened or fully closed. If the air valve is closed, then the chamber pressure is not influenced by the air valve. If the valve is opened then the influence of the chamber pressure is specified using Poiseuille's equation on gasses. The air flow between the air container and the chamber is determined by the pressure difference in the air container and the chamber, the volume of the chamber and the air container, and an air valve constant which depends on factors like the size of the valve. It is possible to open the air valve or close it at any time.

Valve : This process specifies a valve that lets air from the pipeline into the foreline and back. The process $Valve$ in Appendix A specifies a valve that is either fully closed, half open, or fully opened. The mode of the valve in which it is half open is called the softstart mode. Again, we use Poiseuille's equation to specify the pressure change between pipe and foreline. If the pressure in the pipe is higher than in the foreline, air flows from the pipe to the foreline (creating an equilibrium). If the pressure in the foreline is higher than the pressure in the pipe, air flows from the foreline to the pipe. Also this valve can be opened or closed at any time.

Pump: This process specifies the behavior of a pump that pumps air from the chamber to the pipeline. If the pump is turned on, then the pressure in the pipeline decreases. The pressure decrease depends on the speed S_p of the pump. The maximum speed of the pump is modeled by a spline function with sample points taken from the performance curves of a pump manufacturer. This function is omitted from the χ specification in Appendix A for simplicity reasons. The pump speed is expressed in cubic meters of air that can be moved per second. The pump has four modes: *off*, *accelerating*, *braking*, and *on*. If the pump is off, then the speed of the pump is 0. If the pump is accelerating, then the speed increase is modeled by a sigmoid function. This means the acceleration curve is S-shaped. An extra clock variable *clk* is needed to model this curve. This clock variable is reset every time the pump starts accelerating (or braking). If the pump is braking then the speed decrease is modeled by a declining sigmoid curve. If the pump is at full speed it is in the mode *on* and the pump speed is maximal. Because acceleration and braking are modeled by sigmoid curves, the maximum speed and speed 0 are never reached in the specification. If the pump speed comes within a margin ϵ of these limits, then the mode switches from *accelerating* to *on* or from *braking* to *off*. The pump is equipped with a membrane. If the pump is off, then, air may flow from the pipe to the outside world but not the other way around. If the pump is not off, then the pressure decrease in the pipe is modeled by an exponential equation depending on the pump speed S_p and the volume of the pipe. The pump cannot pump down the pipe to a pressure lower than (approximately) $1\ 10^{-4}$ mbar. This value is taken from the pump specification. The pump sends a status message to the controller if the pump reaches the *on* mode or the *off* mode.

Turbo Pump: The *TurboPump* process is similar to the Pump process. The difference is that it operates within a different pressure range and it has a different maximum speed S_{max}. This maximum speed is also modeled by a spline function based on performance curved of a pump manufacturer. The turbo pump does not have a membrane. Therefore, if the pump is off, air flows from the chamber to the foreline or from the foreline to the chamber (depending on the pressure in both foreline and chamber).

7.3.2 The Controller Process

The controller process specifies the pump down sequence and venting sequence. The pumpdown sequence and the venting sequence are specified through a number of modes as defined by the χ language. These modes specify the two sequences as described in the previous section. The controller is specified such that it takes into account the possibility of a sequence being interrupted by another sequence at any moment. If a sequence is interrupted, then the new sequence starts from the beginning. The implementation only performs an output on a change (see next section): e.g. only if the air valve is closed, then the output $airvalve!open$ can be observed. Therefore, the state of the controller (with respect to the hardware) is checked in almost every mode. Notice that the exception is opening the valve in the pumpdown sequence, because in this case the valve can only be in softstart mode.

Initially, the controller process waits to receive a message to start the sequence. The controller assumes that initially the chamber has a stable atmospheric condition.

7.3.3 Composition of Controller and Environment

The processes need to be composed into one model using parallel composition, together with the pressure specifications for the chamber, foreline, pipeline, and the leakage (see Appendix A). All pressure variables are initialized with the value of 1013 mbar (which is 1 atmosphere). The chamber pressure (Cp) consists of the pressure influence caused by the air valve, the turbo pump, and the leaks. It is therefore modeled as: $Cp' = Cp_{av} + Cp_{tp} + Lkp$. The leakage varies depending on the chamber pressure multiplied by some leak constant C_{lk}. This constant models constant factors like the size of the leaks. The foreline pressure Fp is influenced by the turbo pump and the valve: $Fp' = Fp_{tp} + Fp_v$. The pipe pressure is influenced by the pump and the valve: $Pp' = Pp_p + Pp_v$. The send and receive actions are synchronized in channels and encapsulated.

By specifying the channel communications between processes as parameters of the model, these communications are not encapsulated and are viewed can still be viewed by our prototype test tool as separate send actions and receive actions. However, the prototype test tool needs to be adapted such that the tester can indicate which receive actions (of the channels) to consider the input actions for the implementation and which send actions to consider the output of the implementation.

For future test tool implementations it is an option to generate the tests from two

separate models, namely one being the controller and one being the environment. This makes the distinction between a specification of the implementation under test and its environment more explicit.

7.4 Test Setup

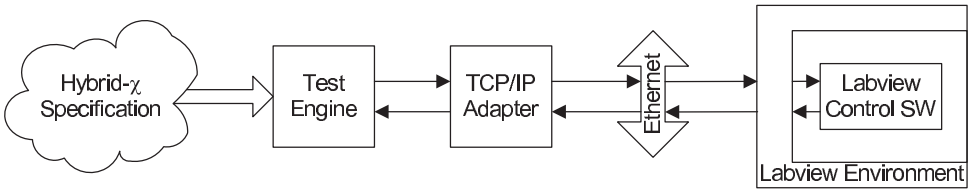

Figure 7.7: NA-Handler Testing

In order to test the vacuum controller we need an adapter and a test environment. The adapter handles the timing of the samples, the communication protocol and infrastructure, and it translates actions, variables, and values between specification and implementation. The Labview test environment implements the interface with the test tool from the implementation side. The adapter and test environment in this case study were built reusing and adapting Frank Stapper's χ to Labview connection. He implemented this connection for co-simulation of the vacuum controller with a discrete-event χ model of the vacuum hardware [45].

The communication between adapter and Labview is established by two socket connections over a regular TCP/IP ethernet network. In order for the Labview controller and the test tool to be able to send and receive messages in parallel two socket connections are used. One socket is used for sending input for the controller from the test tool. The other socket is used for sending output from the controller to the test tool. The Labview controller is the server side of the sockets and the test tool is the client. The implementation has to be initialized first to set up the socket connection. Then, the tests start immediately after the connection with the implementation is setup. Note that to do this the Labview controller is initialized such that it does not perform internal behavior or output before it receives input from the test tool.

The adapter sends input messages and samples over the network and receives output messages and samples from the network. Output is simply passed on to the test-engine. The adapter also takes care of the timing of input messages and samples. The test engine calculates new samples of input and passes them on to the adapter together with the time at which the samples have to be sent to the controller. Input samples are sent to the implementation on the next sample point. This means that the samples need to be computed before this next sample point. We do not take the delay over the network into account. A sample that is sent from the adapter will arrive a fraction of a second later at the implementation. Experiments have shown that in this case study the delay was about 3 milliseconds. Alternatively, under the

assumption that output actions do not occur right before input has to be applied, it is possible to send the input earlier and let the test environment deal with the application of input at the right time.

The adapter also translates actions, types and variables from the χ specification to a format readable for the implementation and the other way around. For instance, in the specification the output channels or of type string, however in the Labview controller the position of the valves is of integer value (0 defines off, 1 defines on, and 2 defines $softstart$).

The test environment receives input messages and samples from the network and sends output messages and samples over the network. Input is simply passed on to the implementation. A time stamp is attached to output messages. This time stamp contains the time that the output message occurred. Inside the test tool this time stamp is used to determine the end of the trajectory of input samples and to determine whether the output action was allowed in the state of the specification. Again, this is only correct, with respect to the test, if there is sufficient time between the output and the next input.

The test environment is implemented such that it only sends an output to the test tool if the controller implementation sends a different message than before. Otherwise, messages from the controller implementation are disregarded.

7.5 Test Results

For testing, besides using less complex differential equations, we needed to reduce the complexity of the model described in Appendix A in some areas. The reason was that our prototype tool implementation needed too much time to compute the sets of allowed actions and trajectories in between two samples. This was caused by the performance of the version of the χ simulator at that time, the performance of the prototype test tool, and the test platform that was used. We ran the test tool on an ordinary 2.6 Ghz Pentium 4 PC with 512 Mb of memory.

First of all, we limited the possibility to interrupt a pump down sequence or a venting sequence. Only after step 6 of the pump down sequence or after step 10 of the venting sequence a new sequence start could be applied as input to the implementation. This meant that we did not need to specify modes for every step in the pump down and venting sequences anymore. Second, the environment processes were removed to reduce the amount of internal actions. Then, automatically the send actions and receive actions of the controller became the observable actions for our test tool again. As a test purpose for input selection we used an alternating ”pumpdown” and ”venting” messages (after a sequence was completed). Third, the pump statuses were modeled by (input) variables with value 0 if the pump was turned off and with value 1 if the pump was turned on. In this way a continuous trajectory did not need to be interrupted by status messages.

With this simplified specification a mistake in a vacuum controller was found (see Figure 7.8).

```
sent: (PMPP1_STATUS,ON) at time 64.9996099472
sent:(TP1_STATUS,OFF) at time 65.9999799728
sent:(CHAMBER_PRESSURE,0.999998) at time 66.000056982
sent: (FORELINE_PRESSURE,1024.0) at time 66.0001199245
sent: (PMPP1_STATUS,ON) at time 66.0001881123
sent: (TP1_STATUS,OFF) at time 66.9995291233
sent: (CHAMBER_PRESSURE,0.899998) at time 66.9996049404
sent: (FORELINE_PRESSURE,1024.0) at time 66.9996681213
sent: (PMPP1_STATUS,ON) at time 66.9997339249
read: (NA_TP1,true)at time 66 with latency 0.999818086624
```

Figure 7.8: Error trace

Figure 7.8 shows the tail of the error trace returned by our prototype test tool. In this test run, the chamber pressure was decreased while the foreline pressure was maintained at 1 bar. This is not a realistic flow. However, since for the pump down sequence only the chamber pressure is relevant for turning on the pumps, maintaining the foreline pressure constant is not a problem.

After 66 seconds (in real time), a turbo pump on output was received with a chamber pressure higher than allowed by the specification. We double checked that indeed the turbo pump should only be turned on at a lower pressure. We verified with the engineer of the vacuum controller that this indeed is not supposed to happen. However, it was not considered a critical error since also due to inaccuracy of the measurements by the sensor the turbo pump probably switches on at higher chamber pressure levels without causing a breakdown of the system.

This proves that hybrid model based testing can be used in industry and can have added value. The mistake that we found in the subsystem controller had not been found by earlier testing by engineers observing the controller behavior during operation, nor in the case study of Frank Stappers [45] on the same vacuum control system. With model checking techniques he found some other issues with the vacuum system , e.g. that according his specification it was possible, however implausible, that the chamber pressure reaches the pressure value of the air container. The reason that this mistake was not found by model checking is that by model checking the correctness of the design is validated, but not the correctness of the (eventual) implementation. This shows that model checking can be used to correct the design but it does not replace the need to validate parts of the real implementation by testing.

8

Conclusions

In this chapter we summarize the results of the work presented in this thesis, we evaluate some aspects of it, and we give some suggestions for future work.

8.1 Results

We defined a formal conformance relation for hybrid systems. This relation defines when a hybrid implementation conforms to a hybrid specification. To our knowledge this is the first conformance relation for hybrid systems. This relation is based on the discrete-event input-output conformance relation by Tretmans [48] and the timed input-output conformance relations by Brandán-Briones and Brinksma [12], and by Krichen and Tripakis [30]. We described another conformance relation that defines whether a hybrid implementation conforms to a hybrid specification with respect to a separate environment. Using an environment model simplifies the formal definition of conformance without changing its meaning.

We have also defined the notion of test for hybrid systems. This notion of test is also based on earlier definitions of discrete-event tests and timed tests. Besides input actions and output actions, continuous input behavior and continuous output behavior is included in the test.

We have prove that the set of hybrid tests derived from a hybrid specification is sound and exhaustive with respect to the conformance relation. This means that if a hybrid test is generated from a specification according to our notion and it leads to the verdict fail, then we know that the implementation under test does not conform to the specification from which the test has been generated. And, if an implementation is not conform to the specification, then we know that a test can be derived from the specification that leads to the verdict fail.

We discussed the issues that arose when we implemented a prototype of a test tool based on our theory. The design of a model-based test tool gave insight in the gap between theory and practice. As a result we adapted our conformance relation and notion of test, replacing real continuous behavior by a notion of sampled behavior.

We have implemented a prototype test tool and experiments have been conducted on two toy examples and in an industrial case study. Part of a vacuum controller was tested against its specification and a mistake was found. This shows that model-

based testing of hybrid systems can be done (admittedly with sampled behavior) and does have added value.

8.2 Evaluation

8.2.1 Applicability in Industry

We have shown that we could find a failure in a hybrid system that was not found in earlier tests nor by modelchecking [45]. Whether hybrid model-based testing will eventually be adopted by industry depends on the benefits it will have. Benefit depends on whether the detected failures weigh up against the extra effort involved in setting up the tests. Assuming that we have a mature implementation, this effort mainly consists of modeling and setting up the connection of the test tool with the implementation.

In order to apply (hybrid) model-based testing in practice, the modeling effort has to be acceptable for industry standards. For a scientific correct approach to model-based testing a formally defined specification language is required. Only in this we know exactly which behavior is tested. The downside for using this kind of languages is that, at the moment, these languages are not commonly used. In order to successfully apply (hybrid) model-based testing in practice time needs to be invested in:

- creating acceptance and willingness to use formal languages;
- training engineers to use the required specification; and languages
- developing easier to use formal languages e.g. based on subsets of existing well known (informal) languages.

In our industrial case study most time was spent on establishing a fast and reliable connection between the test tool and the implementation under test. The generic part of the connection, namely the protocol used for sending information over the TCP/IP network, can be reused in other cases. However, the test environment for the implementation and the part of the adapter will need to be re-implemented with a different implementation. The effort needed for model-based testing is reduced if generic interfaces are available for components to be tested, making it more attractive for industry to use (hybrid) model-based testing.

8.2.2 The Environment Model

In our view an environment model next to the specification leads to a more elegant model for hybrid model-based testing. In our case study it turned out that in order to model our desired input behavior, we needed to model pumps and valves, and even the chamber as separate processes in the specification. However, test generation and execution becomes more complicated in practice since we need to

synchronously take transitions in two models. For our prototype tool, we therefore had chosen not to use an environment model. We recommend that a following case study is undertaken using a separate environment model, and the two approaches are compared.

8.2.3 Sampling

It is arguable whether to still consider using samples of continuous behavior as hybrid model-based testing. Applying and observing samples are discrete events. In this case using a hybrid specification with sampling still has advantages over using a discrete-event model in which the sampled continuous behavior is modeled discrete-event behavior. A hybrid specification is more concise than a similar discrete-event version of the specification. Furthermore, if a hybrid model is used, then the sample rate can be adapted without changing the model. Experiments need to be conducted to investigate the influence of different sampling strategies on the conformance between an implementation and a specification. We have proven that under certain conditions, if an implementation conforms to a specification, then the implementation also conforms to the specification after sampling is applied. It remains an open problem to find the conditions for which we can guarantee that if an implementation conforms to the specification with sampling, it also conforms to the specification without sampling.

8.3 Future Work

8.3.1 Theory Development

There are two issues that are not addressed by the current theory. The first remaining issue is conformance and a notion of test for systems with feedback loops. In these systems the continuous input depends on the continuous output. In order to test the normal behavior of these systems the input described by the test (derived from the specification) has to continuously adapt to the output observed. For instance, if we want to test a thermostat in combination with its heater, then the temperature input provided to the sensor should continuously adapt because the heat coming from the heater continuously influences the temperature. Our notion of test does not allow this because a test prescribes one fixed input behavior over a period of time. A notion of test that may solve the problem could be a notion of test that defines that in a test all trajectories, both on input variables and output variables, allowed by the specification are included. Then, the adaptive input selection depending on the observed output could be handled by the test execution. Unfortunately, such a test does not describe the input behavior in case the output behavior is not allowed by the specification and we do not know what this input behavior should be.

The second remaining issue is to define a conformance relation that defines if an implementation conforms to a specification in case we cannot accurately test whether

an implementation conforms to the specification because of communication delays, clock skews, or rounding. Or, it might be the case that we cannot make an accurate specification but we still want to test whether an implementation conforms to that specification within some limit of deviation. We made the suggestion to use hybrid automata as formal model for the implementation and define a conformance relation that allows deviations on location invariants, flow constraints, and the guards on the switches. However, we do not know whether this satisfactory solves the problem and whether a formal relation can be defined like this. Another option is to maintain the conformance relation, but to create a new notion of test which gives a relative verdict (instead of an absolute verdict **pass** or **fail**). E.g. we can imagine a relative verdict that states that if according to the specification the acceleration of a robot should be 3.0 m/sec^2 and we allow an inaccuracy of 1% and the robot accelerates with 3.01 m/sec^2, then a test leads to a verdict pass with an inaccuracy of $1 - 3.0/3.1 = 0.3\%$.

To develop a theory for relative verdicts we need to define measures for both the possible inaccuracy in continuous behavior and the inaccuracy in discrete event behavior and we need to investigate whether it is still possible to derive tests from a specification based on inaccurate observations, because this behavior is not specified.

8.3.2 Tool Development

In this research we have only built a prototype version of a hybrid test tool. In order to develop model based testing of hybrid systems further, and to make it more applicable for industry, an extensive tool implementation is needed. This tool should include:

- multiple specification language support, which should be easy to use or easy to learn for test engineers;
- an intuitive graphical user interface;
- a fast and reliable test generation and execution algorithm;
- a choice of test selection strategies and stop criteria;
- when sampling is used, a choice of predefined sample rates;
- the possibility to specify custom test purposes and stop criteria;
- when sampling is used, the possibility to define custom sampling strategies;
- comprehensible failure traces; and
- generic adapters, which can be (easily) customized for specific case studies.

8.3.3 Case Studies

More experience with hybrid model-based testing is needed. The first reason is to further develop hybrid model-based testing both in theory and in tooling. The second reason to do more case studies is to show the added value of hybrid model-based testing. This also becomes easier when theory and tooling are developed further.

- Case studies need to be performed that, unlike the vacuum controller, contain continuous output behavior as well. The challenge is this to find an accurate way to observe the (sampled) continuous output behavior of the implementation under test.
- A (prototype) test tool needs to be implemented that uses two models to generate tests, namely a specification and a separate environment. Case studies need to be performed to see whether the addition of an extra model, which increases the computation time for performing tests, is usable in practice.
- Case studies need to be performed using varying sampling rates and sampling strategies to study the effects of sampling.
- Case studies need to be performed to compare timed input-output conformance testing to hybrid input-output conformance testing.
- Case studies need to be performed to create awareness for the use of hybrid model-based testing.

Bibliography

[1] C. Agruss and B. Johnson. Ad hoc software testing - a perspective on exploration and improvisation. `http://http://www.testingcraft.com/ad_hoc_testing.pdf`, 2000.

[2] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. In *Proceedings of EMSOFT'01: First Workshop on Embedded Software*, 2001.

[3] ASML. ASML company webpage. `http://www.asml.com`, 2009.

[4] J. Bach. Exploratory testing explained. `http://www.exploratory-testing.com/articles/et-article.pdf`, 2003.

[5] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. *Journal of Logic and Algebraic Programming, special issue on hybrid systems*, pages 129 – 210, 2006.

[6] A. Belinfante. TorX test tool information. `http://fmt.cs.utwente.nl/tools/torx/introduction.html`, 2009.

[7] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[8] J. Bengtsson and Y. Wang. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3094 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[9] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time applications: Case of planetary rover controller. In *the Workshop Verification and Validation of Model-Based Planning and Scheduling Systems (VVPS'05 - included in the conference ICAPS'05)*, 2005.

[10] H. Bohnenkamp and A. Belinfante. Timed testing with TorX. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Formal Methods Europe 2005*, volume 3582 of *Lecture Notes in Computer Science*, pages 173 – 188. Springer-Verlag, 2005.

[11] M. Bozga, S. Graf, Ileana Ober, Iulian Ober, and J. Sifakis. The IF Toolset. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–268. Springer-Verlag, 2004.

[12] L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *FATESŠ04. Formal Approaches to Testing of Software (4th International Workshop)*, volume 3395 of *Lecture Notes in Computer Science*, pages 64 – 78. Springer Verlag, 2004.

[13] CCITT. CCITT Recommendation Z.120: Message sequence chart (MSC), 2003.

[14] Conformiq Qtronic. Conformiq Qtronic script generation (SG). `http://www.conformiq.com/`, 2009.

[15] P.J.L. Cuijpers, M.A. Reniers, and W.P.M.H. Heemels. Hybrid transition systems. Computer Science Reports 02-12, Technische Universiteit Eindhoven, Department of Computer Science, December 2002.

[16] Côte de Resyste. The côte de resyste project. `http://fmt.cs.utwente.nl/CdR`, 2009.

[17] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES04. Formal Approaches to Testing of Software (4th International Workshop)*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 2004.

[18] G. Frehse. PHAver: Algorithmic verification of hybrid systems past HyTech. In *Proceedings of the Fifth International Workshop on Hybrid Systems: Computation and Control (HSCC)*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer-Verlag, 2005.

[19] H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation and testing. In B. Steffen, editor, *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer-Verlag, 1998.

[20] A. Hartman and K. Nagin. TCBeans software test toolkit. In *Proceedings of the 12th International Software Quality Week*, 1999.

[21] L. Heerink, J. Feenstra, and J. Tretmans. Formal test automation: The conference protocol with PHACT. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems – Procs. of TestCom 2000*, pages 211–220. Kluwer Academic Publishers, 2000.

[22] L. Heerink and J. Tretmans. Refusal Testing for Classes of Transition Systems with Input and Output. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X*, volume 107 of *IFIP Conference Proceedings*, pages 23 – 38. Chapman & Hall, 1997.

[23] T.A. Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Sciences*, pages 265–292. Springer-Verlag, 2000.

[24] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In *Software Tools for Technology*, volume Transfer 1, 1997.

[25] A.T. Hofkamp, R. Meijer, and R.R.H. Schiffelers. Chi 1.0 language and tools. `http://se.wtb.tue.nl/sewiki/chi/start`, 2009.

[26] G.J. Holzmann. Design and validation of computer protocols. Prentice Hall Inc., 1991.

[27] C. S. Horstmann and G. Cornell. *Core Jave 2 - Volume 2: Advanced Features*. Upper Saddle River, 2005.

[28] National Instruments. NI Labview. `http://www.ni.com/labview/`, 2009.

[29] M. Krichen. The timed test generator tool. `http://www-verimag.imag.fr/~krichen/ttg/index.html`, 2009.

[30] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In S. Graf and L. Mournier, editors, *SPIN 2004*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer-Verlag, 2004.

[31] M. Krichen and S. Tripakis. An expressive and implementable formal framework for testing real-time systems. In F. Khendek and R. Dssouli, editors, *17th IFIP Intl, conference on Testing of Communicating Systems (TESTCOM'05)*, volume 3501 of *Lecture Notes in Computer Science*, pages 242 – 257. Springer-Verlag, 2005.

[32] M. Krichen and S. Tripakis. Interesting properties of the real-time conformance relation tioco. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *the 3rd International Colloquium on Theoretical Aspects of Computing - ICTAC 2006*, volume 4281 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[33] K. Larsen, M. Mikucionis B. Nielsen, and A.E. Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306. ACM, 2005.

[34] N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[35] K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. PhD thesis, Technische Universiteit Eindhoven, 2006.

[36] SUN Microsystems. Java SE desktop technologies. `http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp`, 2009.

[37] OMG. Unified modeling language. `http://www.uml.org`, 2009.

[38] M. van Osch. Model-based testing of χ-models. In R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, editors, *Lecture Notes in Computer Science*, volume 3712, pages 227 – 241. Springer-Verlag, 2005.

[39] M. van Osch. Hybrid input-output conformance and test generation. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 70 – 84. Springer Verlag, 2006.

[40] M. van Osch. Hybrid input-output conformance and test generation. CS-Report 06-30, Technische Universiteit Eindhoven, sept 2006.

[41] M. van Osch. Prototype hybrid test tool. `http://www.win.tue.nl/~mvosch/hybridtesting/`, 2009.

[42] PYTHON. Python programming language – the official webpage. `http://www.python.org`, 2009.

[43] J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In F. Cassez and C. Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2008.

[44] Spinroot. PROMELA language reference. `http://spinroot.com/spin/Man/promela.html`, 2009.

[45] F.P.M. Stappers. Modeling, validation, verification and integration of models with the chi-toolkit applied in an ASML case-study. Master's Thesis (confidential), 2007.

[46] L. Tan. Charontester. `http://www.cis.upenn.edu/%7Etanli/tools/charontester.html`, 2009.

[47] L. Tan, J. Kim, I. Lee, and O. Sokolsky. Model-based testing and monitoring for hybrid embedded systems. In *the proceedings of IEEE Internation Conference on Information Reuse and Integration (IRI'04)*. IEEE Society, 2003.

[48] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[49] J. Tretmans and E. Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the First European Conference on Model-Driven Software Engineering*, 2003.

[50] TTCN. TTCN-3. `http://www.ttcn-3.com`, 2009.

[51] Uppsala Universiteit and Aalborg University. UPPAAL TRON. `http://www.cs.aau.dk/~marius/tron/`, 2009.

[52] Verimag. If. `http://www-if.imag.fr`, 2009.

[53] R.G. de Vries. Towards formal test purposes. In J. Tretmans and E. Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01)*, BRICS Notes Series (NS-01-4), pages 61–76, 2001.

[54] R.G. de Vries, A. Belinfante, and J. Feenstra. Testing in practice: The highway tolling system. In I. Schiefendecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems XIV*, pages 210–234, Berlin, Germany, 2002. Kluwer academic publishers.

[55] R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using Spin. *Software Tools for Technology Transfer*, 2(4):382–393, 2000.

A

Vacuum System Specification

This appendix contains the hybrid χ model of the vacuum system as described in chapter 7. Actual values of constants are removed for confidentiality reasons.

The Controller Process

```
proc Controller(Cp, Fp : real, chan ?seq, !airvalve, !valve, !pump, !turbopump, ?pstatus,
                ?tpstatus : string) =
[[ cont  clk : real = 0.0
  var    av, v : string = "closed",  s : string = "manual",
         p, tp, ps, tps : string = "off"
         SOFTSTART_LIMIT : real = ..., TURBOPUMP_LIMIT : real = ...,
         FINAL_VACUUM : real = ..., FINAL_ATM : real = ...,
         UPPERTRESHHOLD : real = ..., LOWERTRESHHOLD : real = ...,
         TIMEOUT : real = ...
:: clk' = 1.0
|| *( pstatus?ps )|| *( tpstatus?tps )
|| [[ mode SEQSTART =
         ( s = "pumpdown" → skip; PD1
         [] s = "venting" → seq?s; SEQSTART
         )
     mode PD1 =
         ( s = "pumpdown" →
           ( av = "opened" → ( airvalve!"close"; av := "closed"; PD2
                             [] seq?s; PD1
                             )
           [] av = "closed" → skip; PD2
           )
         [] s = "venting" → skip; VNT1
         )
```

```
mode PD2 =
   ( s = "pumpdown" →
      ( p = "off" → ( pump!on; p := "on"; PD3
                    [] seq?s; PD2
                    )
      [] p = "on" → skip; PD3
      )
   [] s = "venting" → skip; VNT1
   [] seq?s; PD2
   )
mode PD3 =
   ( s = "pumpdown" →
      ( v ≠ "softstart" → clk := 0.0; ( ps = "on" ∨ clk ≥ TIMEOUT →
                                          valve!"softstart"; v := "softstart"; PD4
                                      [] seq?s; PD3
                                      )
      [] v = "softstart" → skip; PD4
      )
   [] s = "venting" → skip; VNT1
   )
mode PD4 =
   ( s = "pumpdown" →
      ( CP < SOFTSTART_LIMIT → valve!"open"; v := "open"; PD5
      [] seq?s; PD4
      )
   [] s = "venting" → skip; VNT1
   )
mode PD5 =
   ( s = "pumpdown" →
      ( tp = "off" →
         ( CP < TURBOPUMP_LIMIT →
            turbopump!"on"; tp := "on"; PD5
         [] seq?s; PD5
         )
      [] tp = "on" → skip; PD5
      )
   [] s = "venting" → skip; VNT1
   )
```

```
mode VNT1 =
  ( s = "venting" →
    ( tp = "on" → ( turbopump!"off"; tp := "off"; VNT2
                  [] seq?s; VNT1
                  )
    [] tp = "off" → skip; VNT2
    )
  [] s = "pumpdown" → skip; PD1
  )
mode VNT2 =
  ( s = "venting" →
    ( v ≠ "closed" → ( valve!"close"; v := "closed"; VNT3
                     [] seq?s; VNT2
                     )
    [] v = "closed" → skip; VNT3
    )
  [] s = "pumpdown" → skip; PD1
  )
mode VNT3 =
  ( s = "venting" →
    ( p = "on" → ( pump!"off"; p := "off"; VNT4
                 [] seq?s; VNT3
                 )
    [] p = "off" → skip; VNT4
    )
  [] s = "pumpdown" → skip; PD1
  )
mode VNT4 =
  ( s = "venting" →
    ( av = "closed" → ( ps = "off" → airvalve!"open"; av := opened; VNT5
                      [] seq?s; VNT4
                      )
    [] av = "opened" → skip; VNT5
    )
  [] s = "pumpdown" → skip; PD1
  )
mode VNT5 =
  ( s = "venting" →
    ( v ≠ "opened" →
      ( Cp > Fp → valve!"open"; v := "opened"; VNT6
      [] seq?s; VNT5
      )
    [] v = "opened" → skip; VNT6
    )
  [] s = "pumpdown" → skip; PD1
  )
```

```
  mode VNT6 =
    ( s = "venting" →
      ( av = "opened" →
        ( Cp > FINAL_ATM + UPPERTRESHHOLD →
          airvalve!"close"; av := "closed"; VNT7
        [] seq?s; VNT6
        )
      [] av = "closed" → skip; VNT7
      )
    [] s = "pumpdown" → skip; PD0
    )
  mode VNT7 =
    ( s = "venting" →
      ( v ≠ "closed" → ( valve!"close"; v := "closed"; VNT8
                       [] seq?s; VNT7
                       )
      [] v = "closed" → skip; VNT8
      )
    [] s = "pumpdown" → skip; PD1
    )
  mode VNT8 =
    ( s = "venting" →
      ( Cp > FINAL_ATM + LOWERTRESHOLD → skip; VNT5
      [] seq?s; VNT8
      )
    [] s = "pumpdown" → skip; PD1
    )
:: seq?s; SEQSTART
]|
]|
```

The Environment Processes

proc S(chan !seq : string) =
‖[$*($ $seq!$"$pumpdown$"; $seq!$"$venting$" $)$
]‖

proc $AirValve$(cont Cp, Cp_{av} : real, chan ?$airvalve$: string) =
‖[var av : $string$ = "$close$"
:: (av = "$close$" → $Cp'_{av} = 0$
‖ av = "$open$" → $Cp'_{av} = C_{av}.\frac{V_{ac}}{V_c+V_{ac}}.(1013 - Cp)$
)
‖ $*($ $airvalve?av$ $)$
]‖

proc $Valve$(cont Fp, Pp, Fp_v, Pp_v : real, chan ?$valve$: string) =
‖[var v : $string$ = "$close$"
:: (v = "$close$" → $Fp'_v = 0, Pp'_v = 0$
‖ v = "$softstart$" → $Fp'_v = \frac{1}{2}.C_v.\frac{V_f}{V_p+V_f}.(Fp - Pp)$,
$Pp'_v = \frac{1}{2}.C_v.\frac{V_p}{V_p+V_f}.(Pp - Fp)$
‖ v = "$open$" → $Fp'_v = C_v.\frac{V_f}{V_p+V_f}.(Fp - Pp)$,
$Pp'_v = C_v.\frac{V_p}{V_p+V_f}.(Pp - Fp)$
)
‖ $*($ $valve?v$ $)$
]‖

proc $Pump$(cont Pp, Pp_p : real, chan ?$pump$, !$pstatus$: string) =
‖[cont S_p, S_{max}, clk : real = 0,
var p : $String$ = "off", ps : string = "off"
:: $clk' = 1.0$
‖ $S_{max} = \ldots$
‖ (ps = "off" → $S'_p = 0$
‖ ps = "$accelerating$" → $S'_p = clk.(S_{max} - S)$
‖ ps = "$braking$" → $S'_p = -\ clk\ .\ S$
‖ ps = "on" → $S'_p = S_{max}$
)
‖ (ps = "off" ∧ $Fp < 1013$ → $Pp'_p = 0$
‖ ps = "off" ∧ $Fp \geq 1013$ → $Pp'_p = C_p.(1013 - Fp)$
‖ $ps \neq$ "off" → $Pp'_p = \frac{S_p}{V_f}.(1\ 10^{-4} - Fp)$
)

$$
\begin{array}{l}
\| *(\ pump?p \\
\quad ; (\ ps = \text{“}off\text{”} \wedge p = \text{“}on\text{”} \rightarrow ps := \text{“}accelerating\text{”} \\
\quad\quad [] \ ps = \text{“}braking\text{”} \wedge p = \text{“}on\text{”} \rightarrow ps := \text{“}accelerating\text{”} \\
\quad\quad [] \ ps = \text{“}accelerating\text{”} \wedge p = \text{“}off\text{”} \rightarrow ps := \text{“}braking\text{”} \\
\quad\quad [] \ ps = \text{“}on\text{”} \wedge p = \text{“}off\text{”} \rightarrow ps := \text{“}braking\text{”} \\
\quad\quad); \ clk := 0 \\
\quad) \\
\| *(\ (\ ps = \text{“}braking\text{”} \wedge S = \epsilon \rightarrow ps = \text{“}off\text{”} \\
\quad\quad [] \ ps = \text{“}accelerating\text{”} \wedge S = S_{max} - \epsilon \rightarrow ps = \text{“}on\text{”} \\
\quad\quad) \\
\quad\quad ; \ pstatus!!ps \\
\quad) \\
]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{proc}\ TurboPump(\mathsf{cont}\ Cp, Fp, Cp_p, Fp_p : \mathsf{real}, \mathsf{chan}\ ?turbopump, !tpstatus : \mathsf{string}) = \\
[\![\ \mathsf{cont}\ S_{tp}, S_{max}, clk : \mathsf{real} = 0, \\
\quad \mathsf{var}\ tp : String = \text{“}off\text{”}, tps : \mathsf{string} = \text{“}off\text{”} \\
:: \quad clk' = 1.0 \\
\| \ S_{max} = \ldots \\
\| \ (\ tps = \text{“}off\text{”} \rightarrow S'_{tp} = 0 \\
\quad \| \ tps = \text{“}accelerating\text{”} \rightarrow S'_{tp} = clk.(S_{max} - S) \\
\quad \| \ tps = \text{“}braking\text{”} \rightarrow S'_{tp} = - \ clk \ . \ S \\
\quad \| \ tps = \text{“}on\text{”} \rightarrow S'_{tp} = S_{max} \\
\quad) \\
\| \ (\ tps = \text{“}off\text{”} \rightarrow Cp'_p = C_{tp} . \frac{V_c}{V_p + V_c} . (Fp - Cp), \\
\qquad\qquad\qquad\quad Fp'_p = C_{tp} . \frac{V_p}{V_p + V_c} . (Cp - Fp) \\
\quad \| \ tps \neq \text{”}off\text{”} \rightarrow Cp'_p = \frac{S_{tp}}{V_c} . (1 \ 10^{-7} - Cp), \\
\qquad\qquad\qquad\quad Fp'_p = - \ \frac{S_{tp}^2}{V_p} . (1 \ 10^{-7} - Fp) \\
\quad) \\
\| *(\ tpump?p \\
\quad ; (\ tps = \text{“}off\text{”} \wedge p = \text{“}on\text{”} \rightarrow tps := \text{“}accelerating\text{”} \\
\quad\quad [] \ tps = \text{“}braking\text{”} \wedge p = \text{“}on\text{”} \rightarrow tps := \text{“}accelerating\text{”} \\
\quad\quad [] \ tps = \text{“}accelerating\text{”} \wedge p = \text{“}off\text{”} \rightarrow tps := \text{“}braking\text{”} \\
\quad\quad [] \ tps = \text{“}on\text{”} \wedge p = \text{“}off\text{”} \rightarrow tps := \text{“}braking\text{”} \\
\quad\quad); \ clk := 0 \\
\quad) \\
\| *(\ (\ tps = \text{“}braking\text{”} \wedge S = \epsilon \rightarrow ps = \text{“}off\text{”} \\
\quad\quad [] \ tps = \text{“}accelerating\text{”} \wedge S = S_{max} - \epsilon \rightarrow tps = \text{“}on\text{”} \\
\quad\quad) \\
\quad\quad ; \ tpstatus!!ps \\
\quad) \\
]\!]
\end{array}
$$

Model Composition

$$
\begin{array}{l}
\mathsf{model}\ VacuumSystem(\mathsf{chan}\ seq, airvalve, valve, pump, turbopump, \\
\qquad\qquad\qquad\qquad\qquad pstatus, tpstatus : \mathsf{string}\ ) = \\
[\![\ \mathsf{cont}\ Cp, Fp, Pp, Lkp, Cp_{tp}, Cp_{av}, Pp_{tp}, Pp_v, Fp_v, Fp_p : \mathsf{real} = 1013 \\
::\ \ Lkp' = C_{lk}.(1013 - Cp) \\
\|\ Cp' = Cp'_{av} + Cp'_{tp} + Lkp' \ \|\ Fp' = Fp'_{tp} + Fp'_v \ \|\ Pp' = Pp'_v + Pp'_p \\
\|\ AirValve(Cp, Cp_{av}, airvalve) \\
\|\ Valve(Fp, Pp, Fp_v, Pp_v, valve) \\
\|\ Pump(Fp, Fp_p, pump, pstatus) \\
\|\ TurboPump(Cp, Pp, Cp_{tp}, Pp_{tp}, turbopump, tpstatus) \\
\|\ S(seq) \\
\|\ Controller(Cp, Pp, seq, airvalve, valve, pump, tpump, \\
\qquad\qquad pstatus, tpstatus : \mathsf{string}) \\
]\!]
\end{array}
$$

Summary

Automated Model-based Testing of Hybrid Systems

In automated model-based input-output conformance testing, tests are automatically generated from a specification and automatically executed on an implementation. Input is applied to the implementation and output is observed from the implementation. If the observed output is allowed according to the test, then testing may continue, or stop with the verdict *pass*. If the observed output is not allowed according to the test, then testing stops with the verdict *fail*. The advantages of this test method are that:

- specifications can be reused to test every product in exactly the same way,
- test environments can be controlled because the behavior of the environment is specified as the input of the implementation,
- tests can be generated that a test engineer did not think of yet,
- a huge quantity of tests can be generated and repeated endlessly, and
- the test engineer can focus on testing the parts of the system for which tests are not automated.

A hybrid system is a system with both discrete-events and continuous behavior. By continuous behavior we usually mean the behavior of physical quantities over time. A thermostat that observes a chamber temperature and turns on a heater based on the observed temperature change is a system with continuous input and discrete-event output. A robot arm that moves with a certain speed on command (e.g. "GO LEFT") is a system with discrete-event input and continuous output.

Within the TANGRAM project, a four year research project on model-based test and integration methods and their applications, one of the goals was to develop model-based testing for hybrid systems. This involves incorporating continuous behavior and discrete-event behavior into one input-output conformance relation and into a notion of hybrid test. Then, this approach to hybrid model-based testing had to be tried out in practice, in an industrial environment. In this thesis we describe the result of this research.

In Chapter 2 and Chapter 3 we define the necessary preliminaries for defining our conformance relation and notion of test for hybrid systems. We use hybrid transition systems to formally represent the implementation and the specification of a system. We base our conformance relation on the discrete-event input-output conformance relation by Tretmans, and the timed input-output conformance relations by Brandán-Briones and Brinksma, and by Krichen and Tripakis.

In Chapter 4 we define our input-output conformance relation for hybrid systems. In this chapter we also define a notion of test for hybrid systems that we have proven sound and exhaustive with respect to the hybrid conformance relation.

Based on the notion of hybrid test, we have implemented a proof-of-concept hybrid model-based test tool. The architecture of our tool is based on the TorX test tool and the tests are generated from a hybrid specification using the hybrid χ simulation tool. In Chapter 5 we describe TorX and the hybrid χ language.

In Chapter 6 we describe the issues involved in developing a hybrid model-based test tool in general, and our proof-of-concept tool in particular. In order to better fit theory and practice, we adapt our hybrid input-output conformance relation and notion of test to a conformance relation and notion of test for sampled behavior. We have proven that, under certain conditions, if a hybrid implementation conforms to a hybrid specification, then the implementation also conforms to the specification with sampled behavior.

In Chapter 7 we describe the results of a case study that we have performed on a vacuum controller of a waferstepper machine. This controller has sampled continuous input (namely samples of pressure observations) and discrete-event output (namely controlling pumps and valves). We have made a specification that models the sequences of events required for pumping down a vacuum chamber or venting a vacuum chamber. We have modeled the pressure flow in the chamber as continuous behavior. With the proof-of-concept tool we have been able to generate tests, stimulate the vacuum control software with sampled pressure flow, observe output of the vacuum control software, and give a verdict. We have found a fault in the control software that was not found previously in the field, nor by co-simulation of the controller and a model of the hardware, nor by model checking using Uppaal. This result shows that hybrid model-based testing has added value.

In chapter 8 we describe the results of this research and we present some directions for future research.

Samenvatting

Geautomatiseerd Model-gebaseerd input-output conformance testen van hybride systemen

Bij geautomatiseerd model gebaseerd input-output conformance testen, worden tests automatisch afgeleid uit een specificatie en automatisch geëxecuteerd op een implementatie. Input wordt toegepast op de implementatie en output wordt geobserveerd van de implementatie. Als de geobserveerde output geoorloofd is volgens de test, dan kan het testen doorgaan of stoppen met het oordeel *pass*. Als de geobserveerde output niet geoorloofd is volgens de test, dan stopt het testen met het oordeel *fail*. De voordelen van deze testmethode zijn:

- specificaties hergebruikt kunnen worden om elk product op dezelfde manier te testen,
- test omgevingen kunnen onder controle gehouden worden doordat het gedrag van de omgeving gebruikt kan worden als input voor de implementatie,
- er kunnen tests gegenereerd worden waar de test ingenieur nog niet aan had gedacht,
- een grote hoeveelheid tests kan gegenereerd en eindeloos herhaald worden, en
- de test ingenieur kan zich concentreren op het testen van die onderdelen van het systeem waarvoor de tests niet geautomatiseerd zijn.

Een hybride systeem is een systeem met zowel discrete-events als continu gedrag. Een thermostaat die de temperatuur in een kamer observeert en een verwarming aan aanzet afhankelijk van het geobserveerde temperatuurverloop is een systeem met continue invoer en discrete-event uitvoer. Een robot arm die zich op commando (bv. "GA LINKS") voortbeweegt met een bepaalde snelheid, is een systeem met discrete event input en continue output.

Binnen het Tangram project, een vierjarig onderzoeksproject over model gebaseerde test- en integratiemethoden, was één van de doelen om model gebaseerd testen voor hybride systemen te ontwikkelen. Dit bestond uit het samenvoegen van continu gedrag en discrete-event gedrag in één input-output conformance realatie en

een notie van test. Vervolgens, moest deze benadering voor hybride model gebaseerd testen uitgeprobeerd worden in de praktijk, in een industriële omgeving. In dit proefschrift beschrijven we de resultaten van dit onderzoek.

In hoofdstuk 2 en hoofdstuk 3 definiëren we de noodzakelijke voorbereidendingen om onze conformance relatie en notie van test voor hybride systemen te kunnen definiëren.We gebruiken hybride transitiesystemen om formeel de implementatie en de specificatie te representeren. We baseren onze conformance relatie op op de discrete-event input-output conformance relatie van Tretmans, en de input-output conformance relaties met tijd van Brandán-Briones en Brinksma, en van Krichen en Tripakis. Een notie van hybride tests werd gedefinieerd die bewezen sound en exhaustive is in vergelijking met de hybride conformance relatie.

In hoofdstuk 4 definiëren we onze input-output conformance relatie voor hybride systemen. In dit hoofdstuk definiëren we ook onze notie van test. We hebben bewezen dat deze notie van test sound en exhaustive is met betrekking tot de hybride conformance relatie.

Gebaseerd op de notie van hybride test, hebben we een concept hybride test tool geïmplementeerd. De architectuur van deze tool is gebaseerd op de TorX test tool en met behulp van de hybride χ simulatie tool worden tests afgeleid uit een hybride specificatie. In hoofdstuk 5 beschrijven we TorX en de hybride χ taal.

In hoofdstuk 6 beschrijven we de kwesties die betrekking hebben op het ontwikkelen van een hybride model gebaseerde test tool in het algemeen en ons prototype test tool in het bijzonder. Om beter de theory met de praktijk aan te laten sluiten, passen we onze conformance relatie en notie van test aan voor gesampled gedrag. We hebben bewezen dat, onder bepaalde omstandigheden, als een hybride implementatie conformeerd aan een hybride specificatie, dan conformeerd de implementatie ook aan de specificatie met gesampled gedrag.

In hoofdstuk 7 beschrijven we de resultaten van een case studie die we uitgevoerd hebben op een vacuüm controller van een waferstepper machine. Deze controller heeft gesampelde invoer (namelijk druk observatie samples) en discrete-event output (namelijk het besturen van pompen en kleppen). We hebben een specificatie gemaakt die de volgorde van events modelleerde die noodzakelijk zijn om een vacuüm kamer leeg te pompen of te luchten. We hebben het drukverloop in de kamer gemodelleerd als continue gedrag. De concept tool was in staat om tests te genereren, de vacuüm controller te stimuleren met gesampeld drukverloop, de output van de vacuüm control software te observeren, en een oordeel te geven. We hebben een fout gevonden in de control software die nog niet eerder gevonden was gedurende de normale operatie, en die ook niet gevonden was door co-simulatie van de controller en een model van de hardware, en die ook niet gevonden was door model verificatie met Uppaal. Dit resultaat toont aan dat hybride model gebaseerd testen toegevoegde waarde heeft.

In hoofdstuk 8 beschrijven we de conclusies van dit onderzoek en geven we aanwijzigingen voor toekomstig onderzoek.

Curriculum Vitae

Michiel Pieter Willem Jacob van Osch was Born on March 6, 1977, in Deurne, The Netherlands. In 1995 he graduated from Atheneum at the Peelland College in Deurne. Following high school he studied computer science at the Technische Universiteit Eindhoven. In 2001 he graduated after having performed his Master's project at the State University of New-York at Stony-Brook. The subject of this project was modelchecking the CAN bus standard. Following his Master's study he started a Professional Doctorate study in Software Technology at the Technische Universiteit Eindhoven. In 2003 he graduated after a final project at TNO Telecom in Leidschendam. The subject of this project was client-side caching of web pages for PDA applications. After that, he started his Ph.D. track within the TANGRAM project. This was a combined project with the embedded systems institute, ASML, three other universities, and other commercial partners. This project allowed him to use and expand both his experience in theoretical computer science and his experience in software engineering.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Re-*

active Systems. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational*

Complexity of Evolving Systems. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04