**Dissertation**

# Testing of Hybrid Systems using Qualitative Models


Harald Brandl

Graz, 2011

*Institute for Software Technology*
*Graz University of Technology*

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: Prof. Dr. Ivan Bratko
Co-Supervisor: Ass.Prof. Dipl.-Ing. Dr. techn. Bernhard K. Aichernig

# Abstract

Today's software, which runs in an increasing amount of electronic devices, demands efficient testing techniques to meet quality and safety standards. For checking the functional requirements of a system, model-based testing has proven valuable. In the embedded systems domain many devices perform a continuous interaction with their environment. Such systems which show both, discrete and continuous behavior, are called hybrid systems. This thesis deals with the model-based testing of continuous and hybrid systems.

Software failures often occur only under certain environmental conditions. Modeling the continuous system environment allows to generate more meaningful test cases which reflect certain scenarios. Usually, continuous behavior is described with differential equations. This work employs a technique called Qualitative Reasoning to abstract the infinite state space of continuous behavior to a finite state representation. A testing approach based on a new conformance relation is presented which allows to decide the correctness of a continuous system with respect to a given qualitative model. Test cases can be generated due to formal test objectives (test purposes) or a class of coverage criteria.

In order to specify test models for hybrid systems this thesis combines the existing formalism of action systems with qualitative reasoning. By abstracting continuous changes to discrete events it is possible to apply standard input-output conformance testing to hybrid systems. A major question in model-based testing is how to select a finite set from a possibly infinite set of test cases. For hybrid systems this work follows a fault-based test selection strategy. Here, a test case is obtained from the discriminating behavior between an original and a mutated specification. A mutant results from inserting a single fault into a specification. The generated test cases are able to verify if an implementation contains any of the modeled faults or, due to the coupling effect, a class of related faults. Finally, this thesis investigates different approaches to minimize fault-based test suites.

# Zusammenfassung

Die heutige Software, welche in einer zunehmenden Anzahl von elektronischen Geräten vorhanden ist, verlangt nach effizienten Testverfahren um Qualitäts- und Sicherheitsstandards zu erfüllen. Für die Überprüfung der funktionalen Anforderungen an ein System hat sich das modellbasierte Testen als geeignetes Werkzeug erwiesen. Im Embedded-Systems-Bereich betreiben viele Geräte eine kontinuierliche Interaktion mit ihrer Umwelt. Solche Systeme, die sowohl diskretes als auch kontinuierliches Verhalten zeigen, werden hybride Systeme genannt. Diese Arbeit beschäftigt sich mit dem modellbasierten Testen von kontinuierlichen und hybriden Systemen.

Software-Ausfälle treten oft nur unter bestimmten Umgebungsbedingungen auf. Eine Modellierung der kontinuierlichen Systemumgebung ermöglicht es, sinnvollere Testfälle zu generieren, welche bestimmte Szenarien darstellen. Üblicherweise wird kontinuierliches Verhalten mit Differentialgleichungen beschrieben. Diese Arbeit beschäftigt sich mit der Methode des qualitativen Schließens, um den unendlichen Zustandsraum kontinuierlicher Systeme in einen endlichen Zustandsraum zu abstrahieren. Ein Testverfahren, welches auf einer neu entwickelten Konformität basiert, erlaubt es die Korrektheit eines kontinuierlichen Systems in Bezug auf ein bestimmtes qualitatives Modell zu entscheiden. Dabei können Testfälle durch formal definierte Testziele oder einer Klasse von Abdeckungskriterien erzeugt werden.

Um Testmodelle von hybriden Systemen zu spezifizieren, kombiniert diese Arbeit den existierenden Action-Systems-Formalismus mit dem qualitativen Schließen. Durch die Abstraktion von kontinuierlichen Vorgängen zu diskreten Ereignissen wird es möglich, das bekannte input-output Konformitätstesten bei hybriden Systemen anzuwenden. Eine wichtige Frage beim modellbasierten Testen ist, wie man eine endliche Menge aus einer möglicherweise unendlichen Menge von Testfällen auswählt. Für hybride Systeme folgen wir in dieser Arbeit einer fehlerbasierten Testauswahlstrategie. Hierbei ergibt sich ein Testfall aus dem unterschiedlichen Verhalten zwischen einer originalen und einer mutierten Spezifikation. Ein Mutant entsteht durch das Einführen eines Fehlers in einer Spezifikation. Die erzeugten Testfälle sind in der Lage festzustellen, ob eine Implementierung einen der modellierten Fehler oder, aufgrund des Kopplungseffektes, eine Klasse von verwandten Fehlern enthält. Schließlich untersucht diese Arbeit verschiedene Ansätze, um die Anzahl fehlerbasierter Testfälle zu minimieren.

# Acknowledgement

I would like to thank my adviser Franz Wotawa for giving me the opportunity to start my PhD studies on an interesting topic and for his help during the course of this work. I would also like to thank Bernhard Aichernig for his support. He taught me a lot in the area of formal methods and model-based testing.

Further thanks go to my colleagues at the institute. First of all, I would like to mention Willibald Krenn. We shared an office and it was fun to work with him during the course of two projects. I also want to thank Gordon Fraser for his assistance when I was writing my first papers and for helpful discussions via the chat window. Thanks go to Elisabeth Jöbstl for the good teamwork in the Mogentes project and to Martin Weiglhofer and Stefan Galler for interesting discussions. Further thanks go to Stefan Tiran for his collaboration in the Mogentes project.

I want to thank my parents for their support throughout all the years. Finally, thanks go to my former colleagues at FH Joanneum and especially to Hubert Berger who enabled me to work in part-time while starting my PhD studies.

This work was funded by the FIT-IT research project Self Properties in Autonomous Systems (SEPIAS), and the EU project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES).

## Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

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

place, date (signature)

## Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

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

Ort, Datum (Unterschrift)

# Contents

**List of Figures** xiii

**List of Tables** xv

**List of Algorithms** xvii

**1. Introduction** 1

1.1. Model-based Testing . . . . . 1
1.2. Hybrid Systems . . . . . 1
1.3. Qualitative Reasoning . . . . . 3
1.4. Motivation . . . . . 3
1.5. Problem Statement . . . . . 4
1.6. Thesis Statement . . . . . 5
1.7. Research Context . . . . . 5
1.8. Contributions . . . . . 6
1.9. Organization . . . . . 8

**2. Model-based Testing** 9

2.1. Testing Strategies . . . . . 11
2.1.1. Exhaustive Testing . . . . . 11
2.1.2. Testing With Equivalence Classes . . . . . 12
2.1.3. Testing With Purposes . . . . . 13
2.1.4. Random Testing . . . . . 13
2.1.5. Fault-Based Testing . . . . . 14
2.2. Conformance Relations . . . . . 14
2.2.1. Refinement . . . . . 15
2.2.2. Conformance between Labeled Transition Systems . . . . . 17
2.2.3. HIOCO . . . . . 21
2.2.4. RTIOCO . . . . . 21
2.3. Hybrid Systems . . . . . 22
2.3.1. Testing from Hybrid System Models . . . . . 23

2.4. Discussion . . . . . . . . . . 24

**3. Qualitative Reasoning** **25**
3.1. Qualitative Simulation . . . . . . . . . . 27
3.1.1. Discrete Representation of Continuous Change . . . . . . . . . . 27
3.1.2. Qualitative Models . . . . . . . . . . 32
3.1.3. Sign Algebra . . . . . . . . . . 34
3.1.4. Behavior Inference from Qualitative Models . . . . . . . . . . 35
3.2. Simplification of Qualitative Models . . . . . . . . . . 40
3.3. Modeling Continuous Systems with Garp3 . . . . . . . . . . 45
3.4. Qualitative Behavior - A formal Model . . . . . . . . . . 47

**4. Testing of Continuous Systems** **49**
4.1. Conformance between Qualitative Models - qrioconf . . . . . . . . . . 49
4.2. Test Case Selection with Test Purposes . . . . . . . . . . 51
4.3. Coverage-based Test Purposes . . . . . . . . . . 58
4.4. Execution of Qualitative Test Cases . . . . . . . . . . 62
4.4.1. Water Tank – A Continuous System . . . . . . . . . . 62
4.4.2. Mapping between Abstract and Concrete Data . . . . . . . . . . 64
4.4.3. Test Case Execution . . . . . . . . . . 67
4.4.4. Experimental Evaluation . . . . . . . . . . 69

**5. Testing of Hybrid Systems** **73**
5.1. Qualitative Action Systems . . . . . . . . . . 74
5.1.1. Hybrid Modeling . . . . . . . . . . 75
5.1.2. Refinement of Qualitative Actions . . . . . . . . . . 83
5.1.3. Testing . . . . . . . . . . 85
5.2. Automated Conformance Verification of Hybrid Systems . . . . . . . . . . 86
5.2.1. On-the-fly Conformance Checking . . . . . . . . . . 87
5.3. Mutation-based Test Case Generation . . . . . . . . . . 92
5.3.1. Ensuring Controllability in Presence of Non-determinism . . . . . . . . . . 92
5.3.2. Test Case Selection . . . . . . . . . . 94
5.3.3. Experimental Results . . . . . . . . . . 98

**6. Generation of Efficient Test Suites** **101**
6.1. Testing Object-oriented Systems . . . . . . . . . . 102
6.2. Car Alarm System . . . . . . . . . . 103
6.3. Experimental Results . . . . . . . . . . 105
6.3.1. Test Case Generation . . . . . . . . . . 106
6.3.2. Test Case Execution . . . . . . . . . . 109

**7. Conclusions** **113**
7.1. Summary . . . . . . . . . . 113
7.2. Related Research . . . . . . . . . . 115
7.3. Future Work . . . . . . . . . . 116

**List of Acronyms** **120**

**Bibliography** **121**

# List of Figures


1.1. Process of model-based testing. . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2. A hybrid system: the controller operating in its continuous environment. . . . . . 2

2.1. The implementation *i* is not *ioco* to the specification *s*. . . . . . . . . . . . . . . 19
2.2. Hybrid Automaton describing a temperature controller (taken from [83]). . . . . 22

3.1. Models of physical systems and their according behaviors. . . . . . . . . . . . . 27
3.2. (*Diagram bottom, right.*) Function *t_abs* partitions time into equivalence classes. (*Diagram on top.*) Function *v_abs* partitions the range of $f$ into equivalence classes. Equivalence classes are denoted by shaded areas. (*Diagram to the left.*) The resulting qualitative function $q$ is depicted on the left. The qualitative slope is represented by circle, triangle up, and triangle down symbols. Thereby the circle stands for “0”, triangle up denotes “+”, and triangle down represents “-”. . . . . . 31
3.3. QSIM model of the oscillator example. . . . . . . . . . . . . . . . . . . . . . . . 40
3.4. Qualitative cosine oscillation (A), and oscillations (B) and (C) with 90° and 180° phase shift respectively. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.5. Model of the oscillator. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.6. Scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.7. Cycle in the transition system. . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.8. Value history. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

4.1. Conformance between QR transition systems: $I_1$ qrioconf $S$ and $I_2 \not\text{qrioconf}\ S$ . . 51
4.2. SEPIAS Container Tracking Unit. . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.3. Garp3 model of a battery. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.4. Labeled QTS. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5. Minimization of a QTS. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.6. Test case for an empty battery. . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.7. A possible trace through the test case. . . . . . . . . . . . . . . . . . . . . . . 58
4.8. Two-tank system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.9. Qualitative model for controlling the level in the tank with the outlet valve. . . . 61
4.10. Water Tank with two Outlets. . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.11. Hybrid Automaton of the Water Tank. . . . . . . . . . . . . . . . . . . . . . . . 63

4.12. Tank Model Fragment for Mode S0 and S1. . . . . . . . . . . . . . . . . . . . . . 63
4.13. Interleaving of two increasing Quantities. . . . . . . . . . . . . . . . . . . . . . . . 65
4.14. Mapping between Abstract and Concrete Data. . . . . . . . . . . . . . . . . . . . 65
4.15. Landmarks as Real Valued Intervals. . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.16. Slope of observed Values. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.17. Simulink Model of the Water Tank. . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.18. Execution of TC1 on I and M1. . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.19. Execution of TC2 on I and M2. . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

5.1. Two-Tank Pump System. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.2. Example Test Case consisting of four Evolutions. . . . . . . . . . . . . . . . . . 86
5.3. Labeled QAS of the Two-Tank System. . . . . . . . . . . . . . . . . . . . . . . . 87
5.4. Two suspension automata showing the behavior of the water tank example and of a mutated version. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.5. Conformance verification result with a mutated action 'out_pump1_off'. . . . . . 90
5.6. Computation steps of Ulysses. . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.7. Internal choice between an input and an output action. . . . . . . . . . . . . . . . 94
5.8. Product LTS and test case (transitions between shaded states depict the test case). 99
5.9. Example execution of the test case. . . . . . . . . . . . . . . . . . . . . . . . . . 99

6.1. Tool chain for test case generation from UML models. . . . . . . . . . . . . . . 102
6.2. Car Alarm System - state machine. . . . . . . . . . . . . . . . . . . . . . . . . . 103
6.3. Suspension automaton of the Car Alarm System. . . . . . . . . . . . . . . . . . 104
6.4. The two LTSs on the left show parts of the suspension automata of the CAS specification and a mutant. The two LTSs on the right depict the resulting *ioco* product and a selected test case. . . . . . . . . . . . . . . . . . . . . . . . . . . 106

# List of Tables


3.1. Qualitative Addition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.2. Qualitative Multiplication . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.3. Time Point Successors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.4. Time Interval Successors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

4.1. All Test Cases per Test Purpose . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.2. One Test Case per Test Purpose . . . . . . . . . . . . . . . . . . . . . . . . . 62

5.1. Semantics of discrete actions. . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.2. Results when applying conformance verification to mutated specifications. . . . . 98

6.1. Number of generated test cases . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.2. Injected faults into the CAS implementation. . . . . . . . . . . . . . . . . . . 109
6.3. Overview of how many faulty SUTs could not be killed by the test cases generated with different approaches. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
6.4. Number of generated test cases (hand-written OOAS model) . . . . . . . . . . . 110

# List of Algorithms

4.1. Qualitative Test Case . . . . . 56
4.2. execute . . . . . 68

5.1. $getSuccessors(qas, s_1) : L \cup \{\delta\} \mapsto \mathcal{P}(S)$ . . . . . 90
5.2. $getTC(s, Goal) : \mathcal{P}(S \times L \cup \delta \times (S \cup inconc))$ . . . . . 96

# Chapter 1

# Introduction

## 1.1. Model-based Testing

Model-based testing has proven to be a very important instrument to ensure the quality of software [80, 4]. Figure 1.1 illustrates the process of model-based testing. At first, a test engineer builds a formal model from the system requirements which are given in some document format. When the model gets accepted after validation, the testing process can be started. The test case generator takes the model and a test goal, which depends on the selection strategy, as input and produces a set of abstract test cases as output. A collection of test cases is also called test suite. The test execution tool applies all test cases to the system under test (SUT) and gives verdicts about each test run. In terms of software testing the SUT is often referred to as implementation under test (IUT) . During execution the events of abstract test cases have to be translated to events which the SUT understands and vice versa. This is handled by the test adapter. A SUT passes a test suite if all runs in the test results were successful. In this work we deal with the model-based testing of hybrid systems.

In model-based testing different test selection strategies are available. This thesis covers three different approaches:

- Test goals can be formally defined and used to derive tests.
- Test cases can be generated according to coverage criteria. For instance, a set of test cases should cover all states of the model.
- In mutation-based test case generation tests are derived from the discriminating behavior between an original and a mutated specification. This technique is also denoted as fault-based testing.

## 1.2. Hybrid Systems

A hybrid system combines both, discrete and continuous behavior. For instance, a digital controller operating the fuel injection of a car engine builds a hybrid system. Figure 1.2 depicts

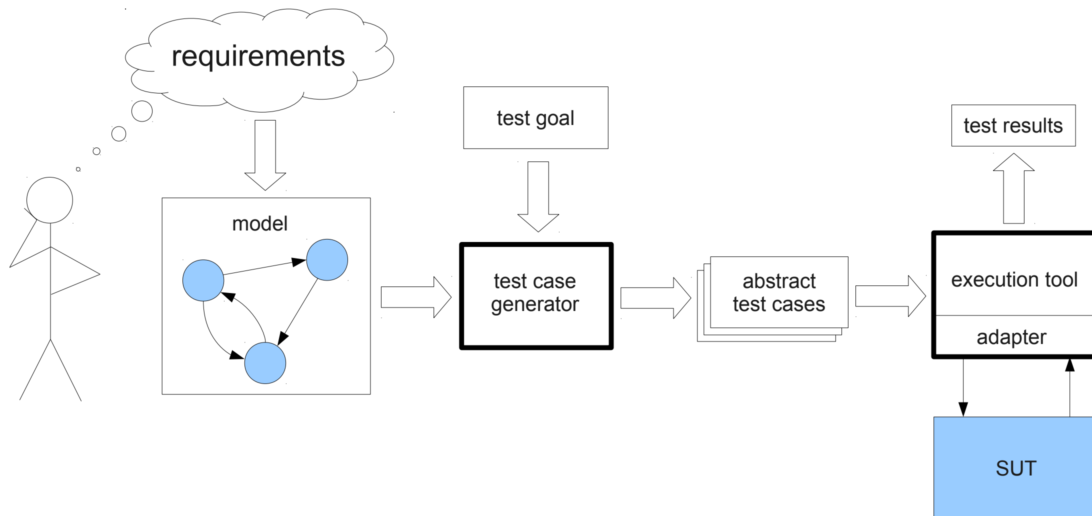



Figure 1.1.: Process of model-based testing.

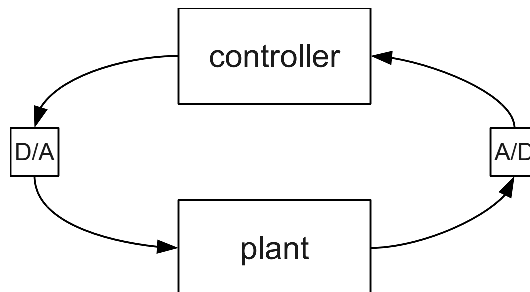



Figure 1.2.: A hybrid system: the controller operating in its continuous environment.

the layout of a hybrid system. The digital outputs of the controller are converted to analog signals which influence processes in the plant, e.g. increasing the speed of a motor. On the other hand measurements from the plant are quantized to digital input values for the controller. The combined operation of controller and plant is called closed-loop behavior. According to control theory the closed-loop view on a system allows to draw important conclusions like stability and long term behavior.

The discrete behavior of the controller is described with computer programs or difference equations. For the continuous behavior differential equations are used. A common formalism to model hybrid systems are hybrid automata by Henzinger [83]. Here, the system state depends on two parts: The discrete state is represented by so-called control locations or modes, i.e. the states of the automaton. The continuous state consists of the valuation of all continuous variables, governed by the differential equations. Section 2.3 shows as an example the hybrid automaton of a temperature controller.

Because of the infinite state space of hybrid systems resulting from the valuation of continuous variables various abstraction techniques exist [32, 33, 119, 90, 152]. This work applies a qualitative abstraction to the continuous dynamics of hybrid systems mainly for two reasons: there exist

general purpose solvers to infer behavior from qualitative models and it is possible to specify generic test models at an user-defined level of abstraction.

## 1.3. Qualitative Reasoning

Qualitative Reasoning (QR) is a technique from artificial intelligence to reason about the behavior of physical systems. The idea is to use an abstract concept of physical knowledge like humans do when they comprehend incidents in the real world. For example, a person knows what happens when throwing a stone in the air without solving any mathematical equations. In QR so-called reasoning engines are able to predict the behavior of a system given a qualitative system description and an initial state.

The theory of QR relies on the abstract interpretation of continuous, time-dependent functions and relations between them. From functions, only their piecewise monotonic behavior is of interest, i.e. if a function is increasing, steady, or decreasing. Furthermore, the range of a function is partitioned into points and adjacent intervals. The points are denoted as landmarks. The domain is mapped to a sequence of time points and time intervals. At each time point a qualitative change happens: either the function changes its qualitative direction or its qualitative value. A value change occurs when the function reaches a point or it enters an interval coming from a point.

In contrast to abstractions with a fixed resolution, qualitative abstraction provides a dynamic resolution which depends on the number of landmarks in the range of a function and the frequency of its qualitative changes. A visualization of this relationship can be found in Figure 3.2.

## 1.4. Motivation

According to Dijkstra "Testing shows the presence, not the absence of bugs" [50]. However, testing increases the confidence in programs and, in contrast to proof techniques, can be fully automated and applied to large systems. When testing is already applied in early development phases the costs of repairing discovered faults are significantly lower than in late phases. In the worst case, faults have to be eliminated in already delivered software which may lead to very high costs or even the end of a company. Today the estimated costs of software testing is about 50% of the overall product costs [126]. Because of this high effort an automation via model-based testing is desirable.

Beside the cost factor software has to be tested thoroughly to ensure safety standards. There have been some well-known software bugs in history which caused threats or injuries to people. For instance from 1985–1987 a bug in the Therac-25 medical accelerator led to injuries and to the death of at least five patients [31]. The system was an improvement over its predecessor by providing two kinds of radiation: a low-power electron beam and X-rays. The root cause of the system failure was a race condition resulting in the configuration of the electron beam in high-power mode without the metal X-ray target being in the right position. Hence, patients were directly exposed to the high-power electron beam.

In many cases software faults lead to the loss or destruction of systems. The first flight of the Ariane 5 on the 4th of June 1996 ended in a disaster [66]. According to the failure report the cause was an error in the Inertial Reference System (SRI). This system was almost identical with the one used in the Ariane 4. The higher speed of Ariane 5 resulted in larger data values within the SRI and finally to an uncaught exception of a data conversion instruction. In particular a 64 Bit floating point number could not be converted to a 16 Bit signed integer number. The exception occurred in a software module which actually serves no purpose after the liftoff of Ariane 5. The flight program could not switch to a second hot-standby system as it had crashed one data cycle before due to the same reason. Based on invalid flight data the control program caused an high attack angle. Self-destruction was triggered about 40 seconds after liftoff when the boosters separated from the main stage.

On the 11th of February 2007 a group of F-22 Raptor jets lost all navigation and communication systems when they crossed the international dateline [56]. The jets had to follow their tankers by visual contact back to the air base. In the case of bad weather this incident would have ended worse.

Mission critical systems in space travel require high testing standards. The work in [120] presents some cases of software problems and analyzes the reasons which led to the damage or loss of space crafts. A further summary and analysis of software related accidents can be found in [166].

Many software bugs only lead to failures when systems get into certain environmental conditions. Therefore, testing should also consider scenarios in the system environment. This thesis deals with the modeling of continuous environments with a technique called *Qualitative Reasoning*. The closed loop view on a discrete controller operating in a continuous environment represents a hybrid system, see Figure 1.2. We present a methodology to generate test cases for continuous and hybrid systems.

Since manual testing is a time consuming task the testing coverage is much lower than can be achieved with model-based approaches. Furthermore, in the manual case, changes in the system specification would require to adapt all existing test cases while in the model-based case a new test suite can be generated merely by pushing a button.

## 1.5. Problem Statement

The challenge of testing continuous and hybrid systems originates from infinite state spaces in the valuation of real-valued variables. This thesis investigates the applicability of Qualitative Reasoning in the domain of model-based testing of continuous and hybrid systems. While there exist various approaches to abstract continuous to finite state behavior most of them rely on numerical information. We follow a purely symbolic approach by abstracting ordinary differential equations to qualitative differential equations. In the qualitative domain real-valued variables correspond to variables having a finite domain of symbolic values.

The behavior of continuous and hybrid systems often depend on many parameters. Sometimes the exact numerical dynamics may not be known or are not reflected in the system requirements.

In such cases Qualitative Reasoning enables the specification of generic system models. When it comes to testing the controller of a hybrid system the approach is well suited to restrict the behavior of a continuous environment. This usually leads to significantly smaller state spaces than testing in unrestricted environments.

The content of this thesis covers the following topics:

**Qualitative Modeling** In order to build a qualitative model of a continuous process it is required to capture its qualitative properties at an appropriate level of abstraction. What are the landmark values of a system where behavior changes from a qualitative point of view?

The size of the model state space mainly depends on the number of variables and the cardinality of their domains. How can the number of variables of a model be reduced while preserving its expressiveness?

**Testing of Continuous Systems** For deciding the correctness of the behavior of a continuous system regarding a given qualitative model a conformance relation is necessary. How can such a conformance relation be applied in the derivation of qualitative test cases? What are the demands on the execution of qualitative test cases?

**Testing of Hybrid Systems** There exist several languages to model hybrid systems. We are interested in a formalism which allows to apply standard input-output conformance testing. This demands the abstraction of continuous changes to discrete events. What is the relation between continuous changes and discrete events? How can we verify the conformance between two given hybrid system models and derive test cases from the discriminating behavior?

**Test Case Selection** Given a formal specification and a conformance relation the question arises which and how many test cases should be selected. Since our models can be nondeterministic, generated test cases have to be adaptive, i.e. they have a branching structure. For testing continuous systems we generate test cases due to test purposes and coverage criteria. How can we formulate test purposes to specify qualitative test cases and what coverage criteria are applicable in this domain?

For the generation of test cases for discrete and hybrid systems we apply a mutation-based approach. How can the size of a test suite be minimized while maintaining mutation coverage?

## 1.6. Thesis Statement

Qualitative reasoning can be applied in model-based testing of continuous and hybrid systems in order to detect a certain class of faults related to the qualitative behavior of such systems.

## 1.7. Research Context

This work was conducted within the projects SEPIAS (Self Properties In Autonomous Systems) and MOGENTES (Model-based Generation of Tests for Dependable Embedded Systems).

SEPIAS was a FIT-IT project in cooperation with Kapsch Carrier Com as industrial partner and had a duration of two years. The aim of SEPIAS was to enable embedded systems to adapt to internal faults and environmental changes while maintaining their services. One result of the project was the development of an approach for testing whether a certain continuous environment conforms to a given qualitative model.

A major question of test case generation is how to select a finite set from a possibly infinite set of test cases. During the MOGENTES project we focused on the test case selection based on fault models. Here, the derived test cases are able to verify if a given system does not contain any of the modeled faults, and by the coupling effect assumption [62], a class of related faults. In the course of this project we developed the *Ulysses* tool which generates test cases for discrete and hybrid systems based on the input-output conformance relation (ioco) [155]. The EU project MOGENTES with a duration of 3 years and 3 month had the following partners:

- the Austrian Institute of Technology (AIT) was the project coordinator,
- Swiss Federal Institute of Technology Zurich / University of Oxford,
- SP Technical Research Institute of Sweden,
- Budapest University of Technology and Economics,
- Ford Forschungszentrum Aachen GmbH,
- Prolan,
- Prover Technology AB,
- Re:Lab S.R.L.,
- and Thales Rail Signalling Solutions GesmbH.

## 1.8. Contributions

The main contributions of this thesis are:

**Qualitative testing theory** We have developed an approach to test continuous systems using qualitative models. The testing theory was implemented in the prototype tool *QRPathfinder* which generates test cases due to test purposes or coverage criteria. Furthermore, the tool enables the execution of test cases and provides an adapter for testing Matlab/Simulink[13] models.

**Qualitative action systems** This thesis presents the formalism of qualitative action systems to specify abstract test models of hybrid systems. In particular, the existing framework of action systems [15, 139] is combined with qualitative reasoning in order to describe the continuous behavior of hybrid systems. A discrete event interpretation of qualitative action systems allows us to apply standard input-output conformance testing to hybrid systems.

**Test case selection** For the mutation-based approach, we have evaluated different test selection strategies. We eliminate mutants, which are already covered by existing test cases. Only for mutants passing this check, new test cases are generated. This leads to smaller test

suites and hence to lower costs in terms of test execution time. Furthermore, we present an algorithm for generating adaptive test cases.

**Testing tool** We have developed the Ulysses tool which is a able to generate test cases for discrete and hybrid systems. It is implemented in SICStus Prolog[1] and is based on the theory of qualitative action systems. The tool follows a fault-based test generation approach. For validation purposes, Ulysses provides an animator which allows to step through a model. Furthermore, Ulysses can generate random test cases for deterministic systems up to a defined length. Test cases and intermediate graphs are stored in a file format which is compatible with the CADP[2] toolbox. This allows to access the CADP tools for model simplification, graph drawing, model checking, and so on.

The following peer reviewed papers were developed during the course of this dissertation:

- The modeling of ordinary differential equations with a tool based on qualitative process theory is presented in Brandl and Wotawa [38]. Furthermore, the work describes a first idea how to derive test sequences from qualitative models.
- In Brandl et al. [40] we discuss the specification of test purposes and Brandl et al. [39] presents the generation of first qualitative test cases.
- For large specifications it may get difficult to formulate sufficiently many test purposes to achieve a certain testing coverage on the model. The work in Brandl et al. [41] deals with the generation of qualitative test cases according to a class of coverage criteria.
- For deciding the correctness of a continuous system regarding a qualitative model a conformance relation is required. The conformance relation *qrioconf* and an approach for executing qualitative test cases are published in Aichernig et al. [8].
- The work in Brandl [37] gives a brief overview about the testing of continuous systems with qualitative models.
- An extension of *action systems* for specifying test models of hybrid systems is presented in Aichernig et al. [6].
- By abstracting continuous changes to discrete events we are able to apply the input-output conformance relation for testing hybrid systems. The paper by Brandl et al. [42] defines a trace semantics of hybrid systems and presents an approach to verify the input-output conformance between two given hybrid models.
- Mutation-based test case generation for hybrid systems is published in Aichernig et al. [5].
- The work in Aichernig et al. [9] deals with the mapping from UML models to action systems and subsequent generation of mutation-based test cases. In order to minimize test suites due to fault coverage we apply different strategies. A case study with a comparison of these strategies and the formal definition of the test case selection algorithm can be found in Aichernig et al. [10].

[1] www.sics.se/sicstus
[2] http://www.inrialpes.fr/vasy/cadp

## 1.9. Organization

The remaining chapters of this thesis are structured as follows: An introduction to the field of model-based testing is given in Chapter 2. Chapter 3 presents the basic concepts of Qualitative Reasoning. After a discussion of qualitative simulation the topic of simplifying qualitative models is treated. Furthermore, the modeling of continuous systems with the tool *Garp3* [44] is discussed, followed by a formal definition of qualitative behavior.

Chapter 4 deals with the testing of continuous systems. First a conformance relation for qualitative models is defined. Then we present the test case generation due to test purposes and coverage criteria. This chapter finishes with the execution of qualitative test cases, demonstrated on an example.

For specifying hybrid systems the formalism of *Qualitative Action Systems* is introduced in Chapter 5. After this we present the conformance verification of hybrid systems which applies in the generation of mutation-based test cases. An algorithm for the selection of adaptive test cases is discussed and demonstrated on an example.

In Chapter 6 we cover the generation of efficient test suites. We present a small case study of testing an object oriented system and evaluate different test selection strategies. The generated test suites are executed on a set of mutated SUTs in order to assess their fault detection capabilities. Finally, Chapter 7 draws conclusions and gives an outlook for future research.

Chapter 2

# Model-based Testing

*Parts of this chapter are taken from 'Model-Based Test Case Generation Techniques: A Survey' [7] which is joint work with Bernhard K. Aichernig, Willibald Krenn, and Rudolf Schlatte.*

Before dealing with model-based testing the question "What is testing?" has to be answered.

> *Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.*

This definition from the IEEE *Software Engineering Body of Knowledge* [34] describes the general purpose of testing. In the domain of software testing a system under test (SUT) or implementation under test (IUT) is examined with a set of test vectors and the observed output behavior is verified against the expectations. Since testing requires the concrete execution of software it is also referred to as dynamic verification. This is in contrast to static verification techniques like model-checking [102] and theorem proving where programs can be proved correct regarding certain properties without executing them. While both techniques aim for the same goal, namely to distinct correct from incorrect programs, the approaches have their strong and weak points.

Static program verification underlies the assumption that all external components to the SUT such as software libraries or other systems behave correct. It can only prove the correctness of the SUT regarding these assumptions. However, if some of the assumptions are invalid in general, because they are too abstract and ignore, e.g., timing aspects, or they do not hold in a concrete environment, e.g., because of an old library version, a program which has been proved to be correct may reveal faulty behavior during its execution. On the other hand a program proof can be quite compact and efficient to compute while exhaustive testing for even small programs is in most cases not possible. The work in [76] elaborates the relation between test and proof and discusses how assumptions can alleviate the process of testing.

The term *model-based testing* refers to a specific subclass in the field of testing methodologies where test cases are derived automatically from a specification written in some formal language. Model-based testing is also denoted as *black-box* testing and verifies the functional behavior of a system. According to the IEEE *Standard Glossary of Software Engineering Terminology* [95] black-box testing is:

> *Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.*

In model-based testing the model serves as *oracle*. During test case execution the observed outputs of the implementation are checked against the oracle, i.e. the specified outputs in the model.

There are two different principles in model-based testing, namely online and offline testing. In online testing the test case is constructed on-the-fly during test execution. Here, the specification and the SUT are executed in parallel while synchronizing on common events. In a certain model state, either the SUT sends an output event which moves the tester to the next state or the tester nondeterministically chooses one input and sends it to the SUT. When the tester cannot follow an output event a fail verdict is issued. Recorded online test cases are not reproducible when dealing with nondeterministic models. Offline test cases are created prior to execution and are not subject to this restriction as the may have a branching structure. Branching test cases are called adaptive because they are able to follow different output decisions of a SUT.

A further distinction is made between active and passive testing. In active testing the tester steers the SUT by sending inputs while reacting to observed outputs. For passive testing the observed behavior of a SUT is replayed on the test model. This process is also called monitoring. In passive testing the inputs for the SUT have to be provided separately as they do not come from the test model. Hence, this approach can be applied to monitor the communication between two entities, i.e. between a system and its user or between two systems.

Other testing approaches are robustness and performance testing. They do not test the functional behavior of the system but test the behavior after unexpected inputs and heavy system loads respectively. Complementary to black-box testing is white-box testing where the internal structure of a system, i.e. the source or machine code, is used to generate test cases that meet certain coverage criteria. Since white-box test cases contain no test oracle manual inspection of executed test cases is required. This work concentrates on black-box testing of discrete and hybrid systems.

Model-based testing has an analogy to natural sciences. For instance, in experimental physics theories are tested by conducting experiments and evaluating the observations. If all observations of sufficiently many experiments can be explained by the theory it is considered as valid. In contrast to this, when in model-based testing an unexpected behavior is observed this indicates that either the model or the implementation is faulty. This has to be resolved by manual inspection of the test run.

In order to denote the cause and effect of wrong behavior in programs the IEEE Standard Glossary of Software Engineering Terminology [95] distinguish between *fault*, *error*, and *failure*. The definition originates in the fault tolerant systems area where a fault is an incorrect program statement or data definition. An error is the difference between the computed value of an operation and the correct, specified value. A faulty program may lead to an error when an incorrect program statement gets executed. Finally, an error can result in a failure when the system cannot perform its required functions anymore.

There exists a lot of literature dealing with model-based testing. The book *Practical Model-*

*based Testing: A Tools Approach* [159] is an introduction to the topic. It gives a motivation why applying model-based testing is worthwhile and presents some common modeling languages like B [1] and UML [141]. Then coverage metrics and test case generation on the presented models is discussed. It also deals with test adaptation, which is the process of making abstract test cases executable, shows two case studies, and elaborates the issues when adopting model-based testing. Furthermore, the following surveys examine different kinds of model-based testing techniques:

- The survey by Hierons et al. [92] presents multiple formal specification languages and discusses their use for model based testing.
- The bibliography by Brinksma and Tretmans [46] and the more recent tutorial by Tretmans [157] contain an overview of the theory, literature and tools of testing based on labeled transition systems, as well as an introduction to model-based testing in general.
- The book by Broy et al. [48] gives an extensive overview of model-based testing of reactive systems. It deals with testing using finite state machines, labeled transition systems, and timed automata. Furthermore, preorder relations and the resulting test case generation techniques are covered.
- Belinfante et al. [24] give an extensive overview of test case generation and execution tools for reactive systems.
- Lee and Yannakakis [118] give a survey of testing techniques using finite state machines; they also deal with related problems such as state identification.
- The survey of Fraser et al. [73] describes test case generation using model checking techniques.
- Gaudel and Le Gall [77] give an overview of theory, tools and case studies of testing against algebraic specifications.

In the next sections we present various testing strategies.

## 2.1. Testing Strategies

There are different strategies that can be used to generate suitable test cases and test suites. In this section, we introduce a classification of testing strategies and discuss their applicability.

Choosing a testing strategy depends on the desired coverage and on the testing assumptions for the SUT. Testing assumptions enable the selection of a finite set of test cases from an infinite state specification in order to check if an implementation is correct regarding the specification. Hence, for a given SUT the assumptions have to hold and it has to pass all test cases. Having no testing assumptions requires exhaustive testing to ensure correctness; the stronger the assumptions the fewer test cases are needed.

### 2.1.1. Exhaustive Testing

According to the IEEE Std 2003–1997, exhaustive testing can be defined as follows:

> Exhaustive testing seeks to verify the behavior of every aspect of an element, including all permutations. For example, exhaustive testing of a given user command would require testing the command with no options, with each option, with each pair of options, and so on, up to every permutation of options. The various command options and permutations rapidly approach numbers too large to reach execution completion in realistic time frame. As an example, there are approximately 37 unique error conditions in POSIX.1. The occurrence of one error can (and often does) affect the proper detection of another error. An exhaustive test of the 37 errors would require not just one test per error but one test per possible permutation of errors. Thus, instead of 37 tests, billions of tests would be needed (2 to the 37th power). Exhaustive testing is normally infeasible [96].

Note, that when dealing with finite models, exhaustive testing may be feasible because of the possibility to enumerate the complete input space. However, even for small programs this is impractical. In most cases, programs show for many input values a similar behavior. If behaviors of different inputs are equal then the inputs form an equivalence class.

### 2.1.2. Testing With Equivalence Classes

The test input data forms *equivalence classes* according to the observations a tester can make. Consider a tester which observes a single output variable and only recognizes whether it is equal to zero or not. Then the input domain can be partitioned into two classes which behave equivalent in terms of the observed outputs. One value from each input equivalence class is sufficient to test all observable output behaviors of the system.

However, such abstract observations require strong testing assumptions that have to be proved. Testing assumptions are formulated as hypotheses, e.g., the uniformity hypothesis by Gaudel [76] states that if the execution of a test case with an input value from an equivalence class leads to a pass verdict it will pass for all input values of this class.

The work by King [108] presents the symbolic execution of programs in the context of testing. The control flow of a program determines the symbolic values of input variables during computation. Each branch taken is recorded in the so-called *path condition*. The path condition determines the equivalence class of input values which lead to the according symbolic state. The exploration of all possible paths through a program yields the symbolic execution tree which may be infinite in the case of looping programs. The work in [105] describes the application of symbolic execution to specifications for the purpose of test case generation.

Symbolic execution has been combined with concrete execution which is referred to as *concolic* or *dynamic symbolic* execution. Here, the path condition during an initial concrete program run is recorded. From this initial path condition new ones can be obtained by negating certain conjuncts. The feasibility of taking a new branch is computed by a constraint solver or an SMT solver like *Yices* [67]. The "flipping" of the path condition provides that different parts of the program can be executed which ensures a better coverage that pure random testing. The work by Godefroid et al. [79] presents the tool DART which relies on dynamic symbolic execution of C programs. Further testing tools and successors of DART are CUTE [147], jCUTE [146] for Java

programs, EXE [51], and Pex [150] for the .Net framework. The work in [81] presents a testing tool for the Creol [100] language.

Notice, that dynamic symbolic execution is a whitebox method as it exploits the source code of a program. However, it can be applied in the context of model-based testing where the test oracle is part of the source code in terms of assertions [70] or contracts [124].

### 2.1.3. Testing With Purposes

A second model-based testing strategy that tries to avoid exponential growth is to use *test purposes*. This method is used to narrow down the specification to some smaller subset before generating test cases. A test purpose can be seen as a slicing criterion that cuts out all parts of the specification which are not of interest for testing. Testing with test purposes opens the possibility of inconclusive test verdicts when the system deviates from the test purpose while staying within the full specification.

A tool using test purposes is the test case generator *TGV* [98], which computes the synchronous product of a test purpose and a specification (both given as labeled transition systems), resulting in a complete test graph (CTG). The CTG contains all test cases satisfying the stated test purpose. Other approaches to testing with purposes use environmental models to restrict specifications. Such models are composed in parallel with the specification, synchronizing on common events. Non-shared events are hidden and disappear, leading to abstract states and possible nondeterminism, but smaller specifications. Further testing tools that allow to state test purposes are SpecExplorer [162] and STG [55].

The tool *UPPAAL* [89] also supports test purposes for the generation of real-time test cases. In addition test cases can be generated randomly or due to coverage criteria. Test purposes and coverage criteria are expressed via so-called *observer automata* [29] which are *Extended Finite State Machines (EFSMs)* [54].

### 2.1.4. Random Testing

Another testing strategy, *random testing*, also tries to avoid the exponential blow-up of exhaustive testing. The strategy employed by the tester is to explore the specification randomly by selecting new input events (and reacting to the output events emitted by the SUT) until a certain bound is reached or a test purpose is satisfied. Most approaches use random testing *on-the-fly* (online) during test case execution rather than precomputing test cases off-line. A random testing tool is *TorX* [158] which works on several specification languages with labeled transition system semantics, e.g. *LOTOS* [97]. The tool *JTorX* [25], implemented in Java, is the successor of *TorX* and is based on a revised version of the conformance relation *ioco* [157] where test cases are input enabled.

When dealing with nondeterministic specifications, online testing has the benefit that test cases are adaptive, i.e., they evolve depending on received outputs from the SUT. On the other hand, online test cases are not always reproducible. A tester may not be able to replay recorded test

cases on nondeterministic implementations. In contrast, an offline test case can handle nondeterminism. Furthermore, many random test cases may be needed to achieve a desired amount of testing coverage.

### 2.1.5. Fault-Based Testing

A fault or mutation is a deviation from a correct program or model. *Mutation testing* can be used for test case generation. The idea of this strategy is to introduce faults in either the specification or the implementation and to generate test cases that discriminate between the original and the, so called, *mutant*. The authors of [99] present a survey from the beginnings of mutation testing to recent works in the field.

Initially, *fault-based testing* or *mutation testing* was applied to assess the quality of a given test suite. The *mutation score* is the ratio of the number of killed mutants over the number of non-equivalent mutants and indicates the effectiveness of a test suite.

Initial work in fault-based testing dates back to the late 1970s (cf. [62]) and, in case of *specification mutation*, the mid 1980s (cf. [49]). In general, fault-based testing is based on two assumptions: the *coupling effect* [62] and the *competent programmer hypothesis* [2]. The coupling effect states that test cases which can detect simple faults are also likely to find more complex faults. The competent programmer hypothesis states that programs are mostly correct. Additionally, the mistakes that are made are often similar, like misnamed variables or wrong conditions in branch statements. Mutation testing uses these assumptions to form *fault models* comprising a set of mutation operators. For example, Black et al. analyzed operators for specification mutation in [28].

Relying on the competent programmer hypothesis, it is possible to anticipate properties of a fault resulting in fault models. Fault-based testing takes advantage of this knowledge: it will construct a mutant that includes the anticipated fault and then search for a discriminating test-case revealing the fault (and hence, via the coupling effect, other faults as well).

The problem of detecting if a mutant is equivalent to the original program (or specification) is undecidable. A mutant is called equivalent if the mutation will not lead to an observable difference between original and mutant. The survey in [99] discusses several approaches to deal with this issue. Offutt and Pan [129, 128] formulate the equivalence problem as constraint satisfaction problem. In our work we follow a similar approach which is related to a technique called *bounded model checking* [27]. In particular we apply bounded conformance verification between a mutated and an original model to recognize equivalent mutants and in the case of non-conformance determine the discriminating behavior, see [42].

## 2.2. Conformance Relations

In formal testing, conformance is defined as a relation between a specification model and an implementation model. If this relation does not hold, an observable failure has been detected. Hence, what is considered a failure is defined by the conformance relation. In order to decide

conformance some testing hypotheses have to be stated [26]. One is that the implementation can be represented with the same formalism as the specification.

A trivial, widely used form of conformance is *observational equivalence*, which demands that a SUT produces exactly the same observations as the reference model. This is the principle of *regression testing*, where new software must behave exactly as its older versions. However, this conformance relation is rather strong – in general a specification model derived from the requirements will be incomplete and should leave implementation freedom for unspecified behavior. Hence, useful conformance relations are preorder-relations rather than equivalence relations, the order going from abstract to more concrete models. In the following, we are going to discuss the most relevant conformance relations.

### 2.2.1. Refinement

Conformance is closely related to the formal notion of *program refinement*. A common test assumption is that the implementation behaves as a formal system whose exact details and structure are unknown [154]. The testing process is then a series of experiments to see whether that unknown formal model conforms to (i.e. is a refinement of) the specification, which is a known formal model.

Refinement stems from the area of program verification and answers the question of whether a program can be safely replaced by a more efficient, refined version. The Vienna Development Method (VDM) advocates refinement as a formal method for developing programs in a step-wise manner from abstract models down to code, with every refinement-step being formally verified [101]. Other formal methods supporting step-wise refinement are RAISE [134] and the B-Method [1]. In addition, refinement calculi have been developed that allow to derive refined programs by following a set of refinement laws [18, 125]. All these refinement techniques are based on a refinement relation defined via the language's semantics.

Refinement can be split into two kinds: *operational refinement* and *data refinement*. Operational refinement is refinement without changing the state-space, e.g. implementing a pre-postcondition contract specified in VDM, RAISE, B, Eiffel, OCL, JML, Spec# or a similar notation [101, 134, 1, 124, 116, 23]. Data refinement maps between programs of different state-spaces, e.g. a balanced-binary tree implementing a set functionality. In the latter, a mapping between the abstract and concrete data is needed, e.g. mapping between binary trees and sets. (For details on data-refinement, we refer to [63].) Such mappings are also applied in model-based testing when abstract stimuli need to be converted to concrete data-formats of the SUT and actual responses, being converted back in order to compare with the abstract expected responses.

Conformance, and hence refinement, depends on observations the formal semantics is defined over. Therefore, different (operational) refinement relations have been proposed for different semantics. In the following, we discuss the most relevant ones.

#### Relational Refinement

Relations between (before-after) states is a standard (denotational) semantics for imperative sequential programs and their abstract contracts. Here, refinement is defined as relation-inclusion,

meaning that a refinement (implementation) does not reach states that are not allowed by the abstract specification. If the state-relations are defined as predicates, refinement is defined as implication from the refined to the abstract. However, this only holds for total relations (specifications). In order to allow for partial specifications, the relations are defined via pre- and postconditions. Then, refinement can be characterized directly via pre- and postconditions: Under refinement preconditions are weakened and postconditions are strengthened. For example, VDM by Jones [101], and Hoare and Jifeng [94] use this kind of refinement for sequential programs.

#### Weakest-Precondition Refinement

An alternative denotational semantics are weakest-preconditions used by Dijkstra and Scholten [64], Back and von Wright [18], and Abrial [1]. Here the syntax of a modeling or programming statement is interpreted as a predicate transformer mapping a given postcondition to the weakest precondition such that the statement will satisfy the postcondition. Refinement is given if and only if the weakest precondition of the refinement is implied by the weakest precondition of its abstract specification. Therefore, compared to the relational model, the implication order is reversed.

#### Axiomatic Refinement

For an algebraic semantics a refinement must satisfy all the axioms of the abstract specification. For example, all implementations of a stack must satisfy the axioms of a stack. The advantage of this form of refinement are the easier equational proofs, which was one of the motivations for adopting this style in RAISE [134].

#### Traces Refinement

For interactive systems which may be non-terminating, different semantic models are needed. Therefore, in CSP [93] additional points of observation are introduced: so called *events* mark the synchronization points between communicating processes where data is exchanged. The semantic domain of such systems are *event traces*, the possible sequences of events of a process. Refinement is then defined as trace inclusion, the traces of the abstract including the ones of the refined process. A consequence of this refinement notion is that the abstract models must be complete: there is no notion of a partial model as expressed in the pre-postcondition style. For partial modeling, a preorder relation like input-output conformance is needed (see Section 2.2.2).

#### Failure Refinement

Traces semantics is not strong enough to express all semantic nuances of a language. For example, it cannot distinguish between internal and external choice of CSP. Therefore, additional refusal sets have been introduced representing the events that cannot be accepted in a certain state. The according refinement extends traces refinement with the additional requirement that

an implementation must only block, if the specification allows blocking. Hence, refinement is extended to negative (blocking) behavior, an idea that has been adopted by the following notion of input-output conformance.

The work of Cavalcanti and Gaudel [53] deals with conformance testing using refusal sets. In particular the authors test for refinement in CSP and discuss issues on test case selection. A comprehensive description of CSP and variants of refinement between CSP models can be found in Roscoe et al. [140].

### 2.2.2. Conformance between Labeled Transition Systems

Labeled transition systems (LTSs) are a common formalism to express the behavior of reactive systems. The semantic of an LTS is defined via event traces which start from an initial state. An LTS can be infinite and nondeterministic and serves as semantic model for specification languages like, e.g., the process algebra LOTOS [97].

**Definition 2.1 (Labeled Transition System (LTS))** *A labeled transition system is a tuple* $M =_{df} (S, L \cup \{\tau\}, \rightarrow, s_0)$*, where S is a countable, non-empty set of states, L is a finite alphabet,* $\tau \notin L$ *is an unobservable action,* $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ *is the transition relation, and* $s_0 \in S$ *is the initial state.*

We use the following common notations:

**Definition 2.2** *Given a labeled transition system* $M = (S, L \cup \{\tau\}, \rightarrow, s_0)$ *and let* $s, s', s_i \in S, a_{(i)} \in L$ *and* $\sigma \in L^*$.

$$
\begin{aligned}
s \xrightarrow{a} s' &=_{df} (s, a, s') \in \rightarrow \\
s \xrightarrow{a} &=_{df} \exists s' \bullet (s, a, s') \in \rightarrow \\
s \stackrel{a}{\not\rightarrow} &=_{df} \nexists s' \bullet (s, a, s') \in \rightarrow \\
s \stackrel{\varepsilon}{\Rightarrow} s' &=_{df} s = s' \vee \exists s_0 \ldots s_n \bullet s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_{n-1} \xrightarrow{\tau} s_n = s' \\
s \stackrel{a}{\Rightarrow} s' &=_{df} \exists s_1, s_2 \bullet s \stackrel{\varepsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\varepsilon}{\Rightarrow} s' \\
s \stackrel{a_1 \ldots a_n}{\Rightarrow} s' &=_{df} \exists s_0, \ldots, s_n \bullet s = s_0 \stackrel{a_1}{\Rightarrow} s_1 \stackrel{a_2}{\Rightarrow} \ldots \stackrel{a_n}{\Rightarrow} s_n = s' \\
s \stackrel{\sigma}{\Rightarrow} &=_{df} \exists s' \bullet s \stackrel{\sigma}{\Rightarrow} s'
\end{aligned}
$$

*Furthermore, for an LTS M we define:*

$$
\begin{aligned}
init(s) &=_{df} \{a \in L \cup \{\tau\} \mid s \xrightarrow{a}\} \\
traces(M) &=_{df} \{\sigma \in L^* \mid s_0 \stackrel{\sigma}{\Rightarrow}\} \\
s \textit{ after } \sigma &=_{df} \{s' \mid s \stackrel{\sigma}{\Rightarrow} s'\}
\end{aligned}
$$

The first relation $init(s)$ defines the set of events enabled in state $s$. The next definition associates to an LTS the according set of event sequences starting from the initial state. The relation *after* determines the set of states reachable after a trace $\sigma$ starting from an initial state. Moreover, an LTS $M$ has finite behavior if all traces have finite length and it is deterministic if $\forall \sigma \in L^* \bullet |s_0 \textit{ after } \sigma| \leq 1$ holds. For conformance relations which employ interaction with inputs and outputs the alphabet $L$ of an LTS is partitioned into input and output labels $L_I$ and $L_U$ respectively.

There exist many conformance relations for labeled transition systems [156]. In the following we present some of the more relevant ones for testing, namely *conf*, *ioconf*, and *ioco*.

#### CONF

In contrast to classical conformance relations like *trace preorder*, which requires inclusion of implementation trace sets in specification trace sets, and *testing preorder* [47], which additionally requires implementation deadlocks to be specification deadlocks as well, the *conf* relation [45, 155] only deals with traces that are part of the specification. This makes it more suitable for testing especially against incomplete specifications, since the large complement of traces that are not in the specification do not need to be considered.

The *conf* relation states that for all traces in the specification the observational behavior of a trace in the implementation, including deadlock behavior, must be a subset of the same trace in the specification.

#### IOCONF

Testing preorder and *conf* are *symmetric* relations which synchronize events of a tester and the implementation, where the models do not differentiate between input and output. In contrast, *input-output relations* deal with models that differentiate between input and output behavior (whereby an output event of the tester synchronizes with an input event of the SUT and vice versa).

The input-output variant of testing preorder is the *input-output testing relation*. The input-output variant of the *conf* relation is called *ioconf* [155]. Informally, *ioconf* requires that for all traces of the specification, the output behavior of the implementation is a subset of the output behavior of the specification.

In the following subsection we discuss *ioco* at length, a more powerful conformance relation than *ioconf* in that it can distinguish traces that differ in time-outs (quiescence) only.

#### IOCO

For the *ioco* relation SUTs are considered to be weak input enabled, i.e. all inputs (possibly preceded by $\tau$ transitions) are enabled in all states: $\forall a \in L_I, \forall s \in S \bullet s \stackrel{a}{\Rightarrow}$. This class of LTS is referred to as $IOTS(L_I,L_U)$ where $IOTS(L_I,L_U) \subset LTS(L_I \cup L_U)$. A state $s$ from which the system cannot proceed without additional inputs from the environment is called *quiescent*, denoted

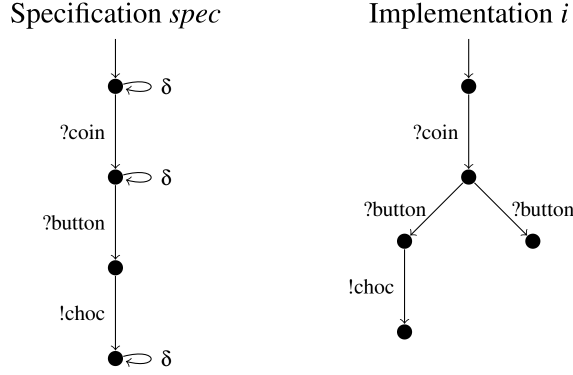



Figure 2.1.: The implementation *i* is not *ioco* to the specification *s*.

as $\delta(s)$. In such a state all output and internal events are disabled: $\forall a \in L_U \cup \tau \bullet s \overset{a}{\not\rightarrow}$. The special label $\delta \notin L$ denotes the absence of any output event in a state. Hence, the transition relation $\rightarrow$ is extended by adding $\delta$ self-loops at quiescent states: $\rightarrow_\delta =_{df} \rightarrow \cup \{(s, \delta, s) \mid s \in S \wedge \delta(s)\}$. Let $M_\delta$ be the LTS over the alphabet $L \cup \{\tau, \delta\}$ resulting from adding $\delta$ self-loops to an LTS $M$. Then the deterministic version of $M_\delta$ is called *suspension automaton* $\Gamma$. The set of suspension traces is

$$Straces(M_\delta) =_{df} \{\sigma \in (L \cup \delta)^* \mid s_0 \overset{\sigma}{\Rightarrow}\}.$$

The following definitions state the set of outputs in a state $s \in S$ and in a set of states $S' \subseteq S$ respectively.

$$out(s) =_{df} \{a \in L_U \mid s \overset{a}{\rightarrow}\} \cup \{\delta \mid \delta(s)\}$$
$$out(S') =_{df} \bigcup_{s \in S'} out(s)$$

Informally, the input-output conformance relation says that for all suspension traces in the specification the outputs of the implementation after such a trace must be included in the set of outputs produced by the specification after the same trace. More formally we define:

**Definition 2.3**
*For implementation models $i \in IOTS(L_I, L_U)$ and specifications $spec \in LTS(L_I \cup L_U)$ the relation ioco is defined as follows:*

$$i\ ioco\ spec =_{df} \forall \sigma \in Straces(spec) \bullet out(i\ after\ \sigma) \subseteq out(spec\ after\ \sigma)$$

A couple of variations of *ioco* have been defined [157]. These variations include definitions of symbolic *ioco* [72], hybrid *ioco* [160], versions of real-time *ioco* [111, 89], and distributed *ioco* [91]. Note, that in difference to [156] in recent work [157] test cases are redefined to be input enabled. This ensures that test cases cannot block outputs from the SUT.

Figure 2.1 shows the specification of a chocolate vending machine, modeled as an input-output labeled transition system that takes a coin (`?coin`) and dispenses a piece of chocolate (`!choc`)

after the user presses a button (`?button`). (We use the common convention of denoting outputs with '!' and inputs with '?'.) It has to be noted that the distinction between inputs and outputs is made from the implementation's point of view. The LTS for the specification is augmented with δ labels (quiescence) in each state where no output event is possible. The deterministic automaton of the augmented LTS is called *suspension automaton*. The specification in Figure 2.1 is a suspension automaton. The implementation *i* shows strange behavior: sometimes the chocolate is delivered as output, sometimes the machine goes quiescent after the button press. This misbehavior is detected, because *ioco* can recognize the absence of the output event – after the two input events *quiescence* (δ "output") is not allowed by the specification, since $out(i \textit{ after } ?coin?button) = \{\delta, !choc\} \nsubseteq \{!choc\} = out(spec \textit{ after } ?coin?button)$.

#### Alternating Simulation

Alternating simulation is introduced in [11] as a refinement relation between alternating transition systems that can be used to model composite systems. Each transition in an alternating transition system corresponds to a possible move in a game between components.

An informal definition of alternating simulation can be found in [60]:

> Consider two systems P and Q. Alternating simulation is a relation between the states of P and the states of Q such that, at related states, all the outputs that can be generated by P can also be generated by Q, and all the inputs that can be accepted by Q can be accepted by P; moreover, corresponding inputs and outputs lead to states of P and Q that are again related.

It was shown in [161] that alternating simulation is equivalent to *ioco* for deterministic systems and input-enabled test cases, which is interesting since in contrast to *ioco*, alternating simulation is not a global property and hence is composable.

Based on alternating simulation, it is possible to define a notion of refinement: Informally, the system P refines Q, if there exists an alternating simulation between the initial states. Having this refinement relation that is based on compatibility of input assumptions and output guarantees, [60] builds the theory for a modeling framework for component-based design and verification.

Microsoft's SpecExplorer [162] builds on Spec# and distinguishes between input (controllable actions) and output (observable actions) for testing. The tool employs alternating simulation as conformance relation.

#### Queued-quiescence Testing

The work in [130] proposes an approach to test input/output transition systems (IOTS) with queues. The approach provides input-enabledness of test cases so that the tester does not need to choose between inputs and outputs. The testing process is separated into two tasks, one generates input sequences and the other one observes according output sequences. The two tasks maintain queues where the SUT consumes inputs from the input process queue and stores according outputs in the output process queue. When the observer process encounters an empty output queue,

recognized as quiescence, it judges about the system's behavior. A system state where the implementation produces no outputs is referred to as *stable*. A trace which transfers the system into a stable state is called *quiescent trace*.

### 2.2.3. HIOCO

The work in [160] introduces the conformance relation *hioco* for hybrid systems, which is similar to the *ioco* testing theory. The conformance relation is defined for *Hybrid Transition Systems* where transitions are labeled with actions that can be discrete or continuous. Continuous actions are called trajectories $\sigma \in \Sigma$ where $\sigma =_{def} (0,t] \rightarrow val(V)$ valuates a set of real-valued variables $V$. Furthermore, the set of actions is partitioned into input and output actions. Note that actions are denoted as events in the context of labeled transition systems.

The *hioco* relation states that for all traces in the specification the following must hold: First the subset inclusion of output actions is defined according to *ioco*. As second condition the trajectories of the implementation must be a subset of the trajectories of the specification after the same system trace. This second subset inclusion is strengthened by filtering trajectories due to specified input trajectories.

The authors propose a test case generation algorithm which needs adjustments in order to be implementable. For instance, the theory deals with infinite state sets since trajectories are defined over dense time and values. In practice one needs to introduce sampling intervals and also deal with the selection of inputs. Furthermore, the theory applies synchronous composition of a test case with an SUT for test case execution. A tester has to process discrete and continuous actions in parallel such that the timing behavior is not influenced by the test driver.

### 2.2.4. RTIOCO

The authors of [89] introduce the definition of *Relativised Timed Conformance* for timed systems. The relation is based on the semantics of timed input/output transition systems (TIOTSs). These automata extend the output actions of untimed input/output transitions with a set of clock variables $d \in \mathbb{R}_{\geq 0}$. Timed automata can be interpreted as extended finite state machines with a set of real-values clocks that can guard the enabledness of transitions. Traces through the system contain beside input and output actions time information about clocks.

The *rtioco* relation assumes non-blocking and weakly input enabled systems/environments. This conformance relation is consistent with the untimed input/output conformance relation *ioco*. It considers traces in the environment to discriminate the output behavior of two TIOTS $S$ and $T$ composed in parallel with their environment $E$. One can see the environment similar to a test purpose used in *TGV* [98] to constrain the behavior of interest. The parallel execution of the specification with the environment yields a set of timed traces; the same applies for the implementation. In *rtioco* the output events and the progress of time are observable, i.e. $Out(s) =_{df} \{a \in A_{out} \cup \mathbb{R}_{\geq 0} \mid s \stackrel{a}{\Rightarrow}\}$ where $s$ is a given state. The relation requires that the observations of the implementation have to be a subset of the observations of the specification after the same environmental trace.

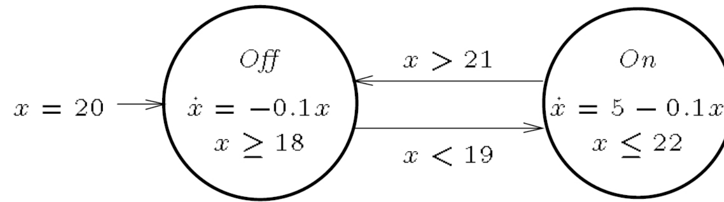



Figure 2.2.: Hybrid Automaton describing a temperature controller (taken from [83]).

The UPPAAL tool [89] provides a model checker for black-box conformance testing of real-time systems. Since real-valued clock variables in timed automata lead to infinite state spaces appropriate abstraction methods are applied, see [65], which enable the application of efficient constraint solving techniques [137]. The tool *UPPAAL cover* [87, 88] implements a test case generation algorithm based on observer automata [29]. For online testing UPPAAL-TRON[113] is available. It provides an API for programming test adapters. An industrial case study for an electronic refrigerator controller can be found in [114].

*Kronos* [36, 59] is another model checking tool for real-time systems. It offers an interface to various real-time formalisms as well as to conventional labeled transition systems.

## 2.3. Hybrid Systems

Hybrid systems combine discrete and continuous behavior. There exist several modeling languages like $\chi$ [122] and CHARON [12] which provide compositional modeling. The work in [139] presents an extension of *action systems* [14] to *hybrid action systems*. The authors describe the parallel composition of two hybrid action systems where global variables are merged and local variables are hidden. The composition is restricted to systems of linear differential equations.

Another formalism for hybrid systems provides the language *Modelica* [74]. It is an object oriented language for modeling hybrid systems with combined continuous and discrete time. It follows the data flow principle and synchronizes continuous flows with discrete time events. In Modelica system components are specified via a set of variables and algebraic equations.

A common formalism for modeling hybrid systems are *hybrid automata* by Henzinger [83]. Hybrid automata are finite state machines where the symbolic states, called *control modes*, represent the continuous evolution of real-valued variables. Automaton states comprise assignments to variables, definition of state invariants, and *flow conditions*. Flow conditions are differential equations that describe the continuous behavior of variables between state transitions. Figure 2.2 depicts the well known example of a temperature controller [83].

The variable $x$ represents the temperature with 20° Celsius as initial value. In control mode *Off* the differential equation $\dot{x} = -0.1 \cdot x$ denotes a flow condition, i.e., the temperature $x$ follows the function $e^{-0.1 \cdot t}$. Further the state contains the invariant $x \geq 18$ declaring that the state is left when the invariant is violated at the latest. The state can be left before as soon as the temperature falls below 19°Celsius. In the *On* control mode a heater causes the temperature to increase. The

invariant of control mode *On* together with the guarded transition to control mode *Off* causes the heater to be turned off somewhere between 21°and 22°.

*Timed automata* are a subset of hybrid automata as they comprise one flow condition per clock variable. The flow condition describes the progress of time, i.e., $\dot{x} = 1$.

Due to infinite state spaces the analysis of hybrid systems is very complex. The work in [85] shows that checking reachability for even a simple class of hybrid systems is undecidable, thus various abstraction techniques have been developed [32, 33, 119, 90, 152]. The tools HYTECH [86] and HYPERTECH [84] are model checkers for hybrid systems.

The work in [131] presents the *Differential Dynamic Logic for Hybrid Systems* and a calculus for proving properties of hybrid systems. The theory is implemented in the hybrid theorem prover *KeYmaera* [132]. The approach has been successfully applied in the verification of a train control system [133] and in the air traffic domain [153].

The work in [90] deals with the modeling of hybrid systems using interval arithmetic constraints. Interval arithmetic provides a means to deal with rounding errors where the real value of a variable is located somewhere within an interval. Systems are specified in the *CLP(F)* language which can state constraints over real numbers and analytic constraints over differentiable functions. The underlying constraint solver calculates, similar to *QSIM* [112], an over approximation of the solution of a system of ODEs. Due to over approximation the solver returns a set of solution intervals. If there is a correct solution to a query it will be in one of the returned intervals. On the other hand not all solutions in the returned set may contain actual solutions. The CLP(F) system solves analytic constraints by using power series to approximate analytic functions. It is also possible to handle non-linear ODEs. A drawback of the approach is the high resource consumption with increasing modeling time scale. This is because of an increase of constraints over Taylor coefficients in the according power series.

Matlab/Simulink[13] is an IDE commonly used in industry for designing and simulating control systems, e.g., in the automotive area. The models can be hybrid containing discrete and continuous parts. When the simulation of a model meets the expectations one can use a code generator like *Realtime Workshop* to get an implementation with certifiable code. Discrete components are modeled with *Simulink Stateflow*. The authors in [52] present an approach to transform time-discrete Simulink models to the data flow language *Lustre* [82].

### 2.3.1. Testing from Hybrid System Models

The authors of [103] propose a test case generation algorithm for hybrid systems and strategies to ensure coverage. Given a testing trajectory the method computes a neighborhood of the initial state of the trajectory. Trajectories with an initial state located inside this neighborhood visit the same control locations of the hybrid system while avoiding unsafe regions. This approach generates test cases based on equivalence classes and ensures coverage on the tested system.

The work in [20] presents an automated test generation approach for hybrid systems with discrete time. As specification formalism *Time Discrete Input-Output Hybrid Systems (TDIOHS)* are used. They build symbolic test cases and in a second step refine them with constraint solving

techniques to executable test cases. The authors applied their tool in the embedded systems domain for testing avionics and railway systems.

The work in [68] deals with randomized test case generation for hybrid systems. Based on the notation of hybrid automata the approach refers to states as $(x, q)$ tuples where $x \in \mathbb{R}^n$ is a valuation of the continuous variables and $q$ is a set of discrete variables which index the system mode. The idea is to explore the state space by building Rapidly Exploring Random Trees (RRTs) [115]. The RRT algorithm has been used in robotics for path planning by computing control signals for trajectories in high dimensional spaces. A RRT consists of nodes (states) which are connected with edges (input actions). For testing, the RRT algorithm is used to find counter examples, i.e. input sequences that drive the system into states that are not in a defined specification set. The authors in [127, 58] extend the random exploration technique with coverage information. Less explored regions are preferred in the exploration process. The usability and performance of RRTs depends on finding appropriate metrics in a system's state space.

## 2.4. Discussion

This chapter presented various testing techniques and conformance relations which are suited for different kinds of system models. Based on the underlying conformance relation certain types of faults can be detected during testing. We are interested in active testing requiring a notion of conformance which considers inputs and outputs. Furthermore, system models should be partial. This means that the specified behavior may describe only a part of the system behavior. Conformance relations like *ioconf* or *ioco* support partial system models as they only consider the behavior after specified traces. For testing hybrid systems the correctness of continuous behavior has to be considered too. From this demands the conformance relation *hioco* would be the right choice to test hybrid systems. However, there are some practical problems in the application of *hioco* since it covers real-valued functions. Thus, we apply the abstraction technique of qualitative reasoning to get a finite state representation of the behavior of continuous and hybrid systems. Based on this abstraction we have developed the Qualitative Reasoning conformance relation *qrioconf* in order to test continuous systems. By giving hybrid systems a discrete event semantics we are able to apply the standard ioco testing theory.

The following chapter gives an introduction to qualitative reasoning.

Chapter 3

# Qualitative Reasoning

*Parts of this chapter have been published in Aichernig et al. [6].*

After explaining the basic concepts of Qualitative Reasoning (QR) this chapter presents our theory to simplify qualitative models. Furthermore, we show how to build qualitative models of differential equations with the QR tool *Garp3*. This chapter concludes with the formal definition of qualitative behavior which serves as our test model.

Qualitative Reasoning is a technique from Artificial Intelligence for reasoning about physical systems with incomplete knowledge. Humans only have limited knowledge about the incidents in the physical world but still are able to cope with them. This kind of common sense reasoning relies on learned, abstract models rather than on detailed, numerical models like differential equations used in physics. Numerical models mostly contain too much information which is not necessary to understand the qualitative properties of a mechanism. Qualitative Reasoning provides a framework to model physical systems and reason about their behavior based on a well-founded theory. Three of the most important works in the area are [109, 71, 112].

The work by De Kleer and Brown [109] presents a qualitative physics based on confluences. Confluences are *Qualitative Differential Equations (QDEs)* which are an abstraction of *Ordinary Differential Equations (ODEs)* . Furthermore, a qualitative calculus is introduced which allows to solve a set of QDEs. Given an initial state and a qualitative model in the form of QDEs, the so-called *envisionment* reflects the qualitative behaviors which may evolve over time. The envisionment is a state graph with possible loops. Each path starting from the initial state represents a qualitative behavior. The work shows the process of qualitative modelling and behavior inference on a water valve example. A central point in qualitative modelling is the *no function in structure (NFIS)* principle. This means that behavioral information of a system should not be incorporated into its description. Following the NFIS principle allows the specify context independent models which can be collected in a library and reused as components to model different systems. For instance, the model of a battery can be applied as component in various models of electric circuits. The NFIS principle has been questioned and considered to be rather a design guideline since it cannot be obeyed in general, see [57, 107]. The main argument against NFIS is the claim

that the context in which a component is used may be required. In certain contexts the structural description of a device may be inadequate in order to infer its function.

Forbus [71] introduces the *Qualitative Process Theory* which refers to the dynamics in a system as a process. The work claims that the theory should provide compositional modeling which supports the NFIS principle by De Kleer. Qualitative variables are denoted as *quantities* and their domains as *quantity spaces*. A quantity space consists of a total ordered set of symbolic values which represent the interesting points where the system's behavior changes from a qualitative point of view. For instance water freezes to ice below 0°C and it is liquid above 0 and below 100°C. All the values below or in between this two points form equivalence classes regarding the qualitative behavior of water. A system is modelled as a collection of *individual views*. An individual view consists of four parts:

1. A set of individuals, i.e. the physical objects that must exist in order to apply the view.
2. Quantity conditions which are inequalities between the quantities of the individuals in the view.
3. Preconditions which must hold.
4. Relations between the quantities which must be true.

In Qualitative Process Theory a view is a process if it contains at least one *influence*. Influences are a means of integration and introduce changes over time in a system. The QR engine *Garp3* by Bredeweg et al. [44] is based on the Qualitative Process Theory and provides a graphical modeling language.

Kuipers [112] followed, similar to De Kleer, a mathematical approach based on QDEs. The work presents the QSIM algorithm for simulating the behavior of qualitative models and shows how to build models of physical systems. Furthermore, dynamic qualitative simulation introduces new landmarks during the simulation of a given system. Additional landmarks do not influence the overall behavior but only the behavior following the state where the landmark was introduced. The dynamic introduction of landmarks allows to draw more conclusions about a system's behavior like increasing, steady, or decreasing oscillations. The work also treats the concept of semi-quantitative simulation. Here, the symbolic real values of landmarks are associated with real-valued intervals. In addition to the model QDEs numerical constraints are stated. These constraints are used to rule out qualitative behaviors for which the additional constraints do not hold.

We started our QR studies with *Garp3* but then recognized that the performance of behavior inference for even small models was not satisfactory. This is because *Garp3* does not support the hiding of model quantities which results in a complete valuation of all quantities and hence in a large number of qualitative behaviors. We then evaluated *QSIM* [112] and used its Prolog implementation *ASIM* [22]. ASIM maps qualitative constraints to integer-arithmetic constraints and employs a constraint solver during qualitative simulation. Furthermore, it supports the hiding of model quantities, applies constraint propagation, and defers the enumeration of variables as long as possible. This provides a very efficient implementation of the QSIM algorithm.

In the following section we give an introduction to qualitative simulation and elaborate the basic concepts of Qualitative Reasoning.

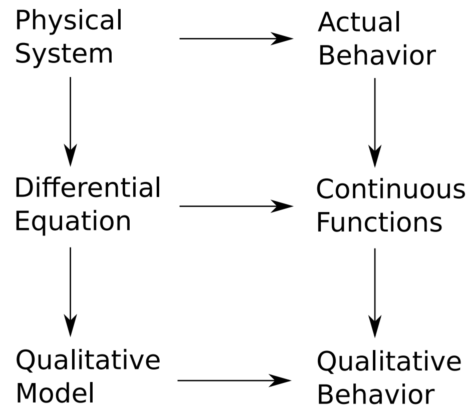



Figure 3.1.: Models of physical systems and their according behaviors.

## 3.1. Qualitative Simulation

Qualitative Reasoning is based on ordinal relations between quantities. According to [112] a quantity is a real-valued attribute of a physical object. *Landmark values denote the "natural joints" that break a continuous set of values into qualitatively distinct regions. A landmark is a symbolic name for a particular real value which might be known or unknown.* The qualitative properties of a real value depend on the ordinal relations to the landmark values. For instance, when we consider the temperature of water from a qualitative point of view we can map the temperature to the domain: below 0°C water is frozen, at 0°C ice melts, above 0°C water is liquid, and around 100°C, depending on the pressure it starts boiling. Above 100°C it changes to steam. This yields the landmark values $AbsZero \ldots Freezing \ldots Boiling \ldots \infty$. Here, we neglect the fact that steam changes somewhere in the interval above the boiling point into plasma. Hence, the distinct values of the quantity water temperature are the landmark values and the open intervals between them.

### 3.1.1. Discrete Representation of Continuous Change

The continuous behavior of a physical system is usually described with differential equations which represent the structure of a system. The variables form the state while mathematical relations between them constrain the possible values the variables can take on over time. A differential equation model of a system can be used to infer the system's behavior in the form of continuous functions of time starting from a given initial state. Such a model allows to make accurate predictions about the behavior of a physical system.

Figure 3.1 shows the relation between physical systems, models, and behaviors in a commuting diagram [112]. No matter what path is followed from the physical system to its continuous functions the result must be the same. This means that the solution of the differential equation model has to match the observations of the system. The lower part of the diagram depicts the abstraction from differential equations to qualitative models. Here, the numerical information is abstracted to symbolic values. If the qualitative model is a valid abstraction of the differential equation model

then it is guaranteed that the according qualitative behavior will cover the qualitative behavior of the continuous functions.

By having equations over symbolic values one is able to model a system where the exact numerical is not known. The equations are called Qualitative Differential Equations (QDEs) and are an abstraction of ODEs.

**Definition 3.1** *According to [112] a QDE is defined as a tuple* $(Q, QS, C, T)$ *where*

- $Q$ is a set of *variables*, each of which is a "reasonable" function of time.
- $QS$ is a set of *quantity spaces*, one for each variable in $Q$.
- $C$ is a set of *constraints* applying to the variables in $Q$. Each variable in $Q$ must appear in some constraint.
- $T$ is a set of *transitions*, which are rules defining the boundary of the domain of applicability of the QDE.

In contrast to [112] we denote the set of variables and quantity spaces with $Q$ and $QS$ respectively. Qualitative variables $Q$ represent time-varying quantities. In order to be compatible with the notation of Qualitative Process Theory we refer to qualitative variables as *quantities*.

In order to express a time-dependent function with a finite set of symbols we have to define a link between these two domains. As stated in Definition 3.1 such functions $f(t)$ have to be reasonable. This means, that $f(t)$ must be continuously differentiable, i.e. $f(t)$ and its first derivation $f'(t)$ are continuous. We denote the set of functions satisfying this property with $C1$. Furthermore, we assume functions to be defined over the extended reals $\mathbb{R}^*$ which include the endpoints $-\infty$ and $\infty$. A function $f : [0, \infty] \mapsto \mathbb{R}^*$ is considered to be continuous at $\infty$ iff $\lim_{t \to \infty} f(t)$ exists. By using the extended reals it is ensured that the value of a variable is always enclosed between landmark values (or lies on a landmark value). The consideration of infinite time intervals allows to reason about the asymptotic behavior of a function. For instance $e^t$ and $e^{-t}$ are reasonable functions on $[0, \infty]$, but *sin* $t$ is not. A further constraint on reasonable functions is that they do not change infinitely often in a finite time interval. For example, *sin* $1/t$ is continuous differentiable but is not reasonable since it oscillates infinitely quickly around $t = 0$.

For capturing the qualitative behavior of a reasonable function only a few landmark values are required. According to [112] a quantity space is a finite, totally ordered set of landmarks values, i.e. $l_0 < l_1 < \ldots < l_k$. A landmark is a symbolic name for a certain value in $\mathbb{R}^*$ which might be unknown. The QSIM tool supports the dynamic creation of additional landmarks for critical values of $f(t)$, i.e. where $f'(t) = 0$. When using the so-called *envisionment* simulation no additional landmarks are created. This has as consequence that critical points of a function may lie between landmarks. Thus, there are fewer distinct states in simulated behaviors. In the following we consider the implicit, open intervals between landmarks as part of a quantity space forming an ordered set $\langle l_0, l_0..l_1, l_1, \ldots, l_{k-1}..l_k, l_k \rangle$.

Time is represented with the qualitative variable *time* which has the quantity space $t_0 < t_1 < \ldots < t_n < \infty$. A time point $t \in [a, b]$ is a distinguished landmark time point of a reasonable function $f$ if it has a boundary element of the set

$$\{t \in [a, b] \mid f(t) = x, \textit{where } x \in \mathbb{R}^* \textit{ is represented by a landmark value of } f\}$$

, see [112]. If a function remains constant at a landmark value for a certain time interval only the endpoints of such an interval are landmark time-points. The qualitative values of a function $f$ over time regarding a given quantity space $QS_q$ are $(qmag, qdir)$ tuples. Here, $qmag \in QS_q$ is either a landmark value $l_j$ if $f(t) = l_j$ or is an interval value $l_j..l_{j+1}$ if $f(t) \in (l_j, l_{j+1})$. The qualitative gradient $qdir \in \delta$ is defined as follows:

$$\delta =_{df} \begin{cases} + & \textit{if } f'(t) > 0 \\ 0 & \textit{if } f'(t) = 0 \\ - & \textit{if } f'(t) < 0. \end{cases}$$

Here, the special values $+$ and $-$ denote the intervals $(0,\infty)$ and $(-\infty,0)$ respectively. Because quantity spaces form a strict total order, we can define an indexing function $ind_q : QS_q \mapsto \{i \mid 0 \leq i < |QS_q|\}$ that returns the index $i \in \mathbb{N}_0$ of a given value from a quantity space $QS_q$.

In [6] we describe the abstraction from continuous functions to qualitative behaviors (traces):

$$\alpha_q : C^1 \mapsto (\mathbb{N}_0 \overset{\sim}{\mapsto} QS_q \times \delta).$$

The abstraction function $\alpha_q$ maps a continuous function $f$ to a qualitative trace $q$, i.e. $\alpha_q.f = q$. The abstraction process is two-fold: (1) concrete real values are mapped to qualitative values and (2) continuous time is mapped to a sequence of states. For a time-dependent function $f : C^1$, iterated application with progressing time values will give a trajectory, i.e. a trace through the range of this function. Given such a trajectory, we use a value abstraction function $v_abs_q : C^1 \mapsto (\mathbb{R}_0^+ \overset{\sim}{\mapsto} QS_q \times \delta)$ that maps quantitative values into the qualitative domain and a time abstraction function $t_abs_q : C^1 \mapsto (\mathbb{R}_0^+ \mapsto \mathbb{N}_0)$ that maps continuous time to discrete states to derive the qualitative trace $q : \mathbb{N}_0 \overset{\sim}{\mapsto} QS_q \times \delta$.

We define the abstraction from continuous values to qualitative values as follows:

**Definition 3.2 (Value Abstraction)** *Given a continuous function $f : C^1$, its corresponding quantity space $QS_q$, and a value abstraction function*

$$v_abs_q : C^1 \mapsto (\mathbb{R}_0^+ \overset{\sim}{\mapsto} QS_q \times \delta).$$

*For each concrete value $f.t$ in the range of the continuous function $f$ the corresponding qualitative value is calculated by the function application $v_abs_q(f)(t)$.*

Note, that our $v_abs_q$ results in a partial mapping modeling the case where the abstract quantity space (landmarks, intervals) does not cover the full range of $f$. For example, given a quantity space $\langle 0, max\rangle$ with $max = 10$ and the function $f$ exceeding this maximum, i.e. $\exists t \in dom(f) \bullet f(t) > 10$, the abstraction is undefined. Here, $dom$ gives the domain of a function. Therefore, in QR special landmarks covering the border intervals up to $\pm\infty$ are usually added. Hence in the following, we assume $v_abs_q$ being total.

Value abstraction is necessary but not sufficient: abstracting from time is also needed for our qualitative abstraction mapping $\alpha_q$.

**Definition 3.3 (Qualitative Abstraction)** *The abstraction $\alpha_q$ is a mapping of continuous time-dependent functions $f : \mathbb{R}_0^+ \mapsto \mathbb{R}^*$ to qualitative traces $q : \mathbb{N}_0 \mathrel{\tilde{\mapsto}} QS_q \times \delta$ such that:*

$$\forall f : C^1, \forall t : \mathbb{R}_0^+, \exists s : \mathbb{N}_0 \bullet \alpha_q(f)(s) = q(s) = v_abs_q(f)(t) \tag{3.1}$$

*Furthermore, a state and its successor must not have equal values:*

$$\forall s_1, s_2 \in dom(q) \bullet s_2 = s_1 + 1 \implies q(s_1) \neq q(s_2) \tag{3.2}$$

**Corollary 3.1 (Time Abstraction)** *By skolemization of the existential quantifier in (3.1), we introduce a function $t_abs_q : C^1 \mapsto (\mathbb{R}_0^+ \mapsto \mathbb{N}_0)$ partitioning the domain of $f : \mathbb{R}_0^+ \mapsto \mathbb{R}^*$ into qualitative equivalence classes:*

$$\forall f : C^1, \forall t : \mathbb{R}_0^+ \bullet \alpha_q(f)(t_abs_q(f)(t)) = q(t_abs_q(f)(t)) = v_abs_q(f)(t)$$
$$= \forall f : C^1 \bullet \alpha_q(f) \circ t_abs_q(f) = q \circ t_abs_q(f) = v_abs_q(f)$$

*This mapping $t_abs_q$ represents our time abstraction.*

Furthermore, function *t_abs* has the following properties:

$$\forall f : C^1, \forall t_1, t_2 : \mathbb{R}_0^+ \bullet t_1 < t_2 \implies t_abs_q(f)(t_1) \leq t_abs_q(f)(t_2) \tag{3.3}$$

$$\forall f : C^1, \forall t : \mathbb{R}^+, \exists \varepsilon > 0 : \mathbb{R} \bullet t_abs_q(f)(t) - t_abs_q(f)(t - \varepsilon) \leq 1 \tag{3.4}$$

$$t_abs_q(f)(0) = 0 \tag{3.5}$$

The properties basically say that *t_abs* has to be increasing over time and that it must not step-over a state, in other words, it has to sequentially visit all numbers $\in \mathbb{N}_0$ up to the current value. A property of time abstraction is the fact, that finite trajectories result in finite qualitative traces but the reverse may not be true. For instance the finite trace $\langle (0,+), (0..max,+), (max,0) \rangle$ may be refined into an infinite exponential function which has the landmark *max* as limit value: $\lim_{t\to\infty} max \cdot (1 - e^{-t}) = max$.

In practice we may not have access to the function definition of $f$ but we may have access to samples of $f$. The well known sampling theorem describes conditions under which $f$ can be reconstructed from samples. Similarly, given a sampling interval $T_s > 0 : \mathbb{R}$, a sample number $t_{\sqcap\!\sqcup}$, and a trace of abstracted samples $\langle v_abs(f)(0 \cdot T_s), \ldots, v_abs(f)(t_{\sqcap\!\sqcup} \cdot T_s), \ldots \rangle$, it is not always possible to reconstruct an abstract qualitative function $q$ from these qualitative values: Within a qualitative function $q$, the values may change to the next value of the quantity space or the next value of the qualitative derivation in one discrete time-step. The following definition, in which $v_1$ and $v_2$ denote qualitative values, states this property formally.

**Definition 3.4 (Continuity of Qualitative Samples)**

$$\exists T_s > 0 : \mathbb{R}, \forall 0 \leq \varepsilon < T_s, \forall t_{\sqcap\!\sqcup} : \mathbb{N}_0, \exists v_1, v_2 : QS_q \times \delta \bullet$$
$$v_abs_q(f)(t_{\sqcap\!\sqcup} \cdot T_s + \varepsilon) = v_1 \wedge v_abs_q(f)((t_{\sqcap\!\sqcup} + 1) \cdot T_s + \varepsilon) = v_2 \wedge Cont(v_1, v_2)$$

*where*

$$Cont((qmag_1, qdir_1), (qmag_2, qdir_2)) =_{df}$$
$$|ind_q(qmag_1) - ind_q(qmag_2)| \leq 1 \wedge |ind_\delta(qdir_1) - ind_\delta(qdir_2)| \leq 1$$

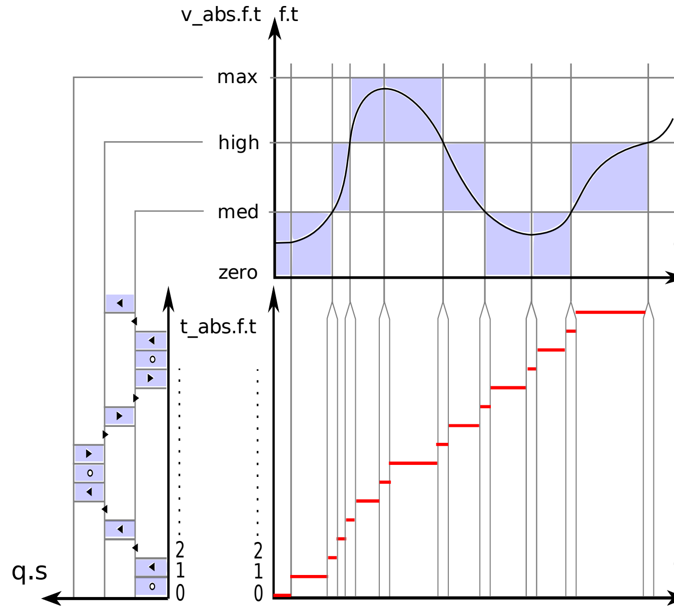



Figure 3.2.: (*Diagram bottom, right.*) Function *t_abs* partitions time into equivalence classes. (*Diagram on top.*) Function *v_abs* partitions the range of $f$ into equivalence classes. Equivalence classes are denoted by shaded areas. (*Diagram to the left.*) The resulting qualitative function $q$ is depicted on the left. The qualitative slope is represented by circle, triangle up, and triangle down symbols. Thereby the circle stands for "0", triangle up denotes "+", and triangle down represents "-".

**Theorem 3.1** *Whenever a system conforms to Definition 3.4 ,we are able to compute a qualitative function q that represents the continuous function f out of the observed sample values.*

**Proof 3.1.** Let us assume that the sampling interval $T_S$ approaches zero. Then discrete time changes to continuous time which leads to a complete qualitative abstraction $q$ of the continuous function $f$. Depending on the change rate of $f$ the sampling interval can be increased up to a critical value where the continuity law still holds. □

In practice the choice of the sampling interval should result in a high enough oversampling of $f$ such that the fastest signal changes of interest can be captured.

Figure 3.2 shows a sketch of the abstraction of a given continuous function $f$ according to four landmarks, i.e. $\langle zero, med, high, max \rangle$. Therefor, we have to associate each landmark with a numerical value. In the upper-right diagram the domain of $f$ is partitioned into three intervals. The vertical lines denote changes in the qualitative behavior of $f$ obtained by applying the value abstraction function *v_abs*. For example, at time point zero the output of the continuous function is below the landmark *med* but above the landmark *zero*, hence in the interval *zero..med*, and the slope of the function is zero. The second abstraction we need to employ in order to map a continuous function to a qualitative one is time-abstraction (*t_abs*). Starting again at the upper-right diagram in Figure 3.2 we can see the operation of time abstraction in the diagram below (bottom right). Informally stated, as long as value abstraction returns the same qualitative value

over passing time on a continuous function, the "qualitative time" is the same, i.e. there is no need to create a new qualitative state within the qualitative function that is being constructed. Taking value abstraction and time abstraction together, we get the resulting qualitative function $q$ depicted in the diagram on the bottom to the left.

According to [112] a qualitative behavior of a continuous function $f$ on an interval $[a,b]$ is a sequence of qualitative values $QV(f,t_0), QV(f,t_0,t_1), QV(f,t_1), \ldots, QV(f,t_{n-1},t_n), QF(f,t_n)$, which alternate between values at time points and values on intervals between time points. The time points and time intervals of a qualitative behavior are referred to as states. Furthermore, a system is a set $F = \{f_1, \ldots, f_n\}$ of reasonable functions $f_i : [a,b] \mapsto R^*$ where each function has its own set of landmarks and distinct time points. The qualitative behavior of the system consists of states where the distinguished time points are the union of the time points of the individual functions. The values of qualitative states are $n$-tuples of the values of the individual functions:

$$QS(F,t_i) = (QV(f_1,t_i), \ldots, QV(f_n,t_i))$$
$$QS(F,t_i,t_{i+1}) = (QV(f_1,t_i,t_{i+1}), \ldots, QV(f_n,t_i,t_{i+1}))$$

If $t_i$ is not a distinct time point of $f_j$, then it must be contained within an interval $(t_k,t_{k+1})$ of $f_j$. Hence, there occurs no qualitative change at $t_i$, i.e. $QV(f_j,t_k,t_i) = QV(f_j,t_i) = QV(f_j,t_i,t_{k+1}) = QV(f_j,t_k,t_{k+1})$. Since at each transition from a time point to an interval or from an interval to a time point at least one function (quantity) changes its value there are no two successive states with the same valuation, see Definition (3.3). The progress of time is represented as a sequence of states alternating between time points and intervals.

### 3.1.2. Qualitative Models

Commonly, ordinary differential equations are used to specify continuous behavior instead of continuous functions. Similar to ordinary differential equations in the continuous domain, in QR qualitative differential equations (QDEs) are used to describe qualitative behavior. QDEs are composed of a set of qualitative constraints, see Definition 3.1, which state relations between model variables (quantities). The *QSIM* tool is the reference implementation of the Qualitative Simulation [112] and in the following we present the most important types of constraints. A complete list can be found in the user manual [69].

$$(add\ x\ y\ z) =_{df} x(t) + y(t) = z(t)$$
$$(mult\ x\ y\ z) =_{df} x(t) \cdot y(t) = z(t)$$
$$(minus\ x\ y) =_{df} x(t) = -y(t)$$
$$(d/dt\ x\ y) =_{df} \frac{d}{dt}x(t) = y(t)$$
$$(constant\ x) =_{df} \frac{d}{dt}x(t) = 0$$

For expressing functional constraints QSIM provides a monotonic function relation:

$$(M^+\ x\ y) =_{df} y(t) = f(x(t)),\ f \in M^+ \tag{3.6}$$
$$(M^-\ x\ y) =_{df} y(t) = -f(x(t)),\ f \in M^+ \tag{3.7}$$

where $M^+$ denotes the set of monotonic increasing functions $f : [a,b] \mapsto \mathbb{R}^*$ with $f' > 0$ over $(a,b)$. The monotonic function relation states that the functions $x$ and $y$ both have the same direction of change in the interior of the domain, i.e. $(a,b)$, of the mapping function $f$, For example a function $y(t) = x(t)^2 + 3x(t)$ gets $(M^+\ x\ y)$ since $y'(t) = 2x(t) + 3 \wedge \forall t \geq 0 \bullet y'(t) > 0$. We are able to model continuous systems by mapping ODEs to qualitative constraints. The mapping process is called structural abstraction where a given ODE is decomposed into a set of simultaneous equations which are then mapped to qualitative constraints. Any ODE $F(y, y', \ldots, y^n) = 0$ of order $n$ can be rewritten into a system of $n$ first-order differential equations:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ &\vdots \\ y_{n-1}' &= y_n \\ y_n' &= F(y_1, \ldots, y_n) \end{aligned}$$

where $y_i := y^{(i-1)}$.

Let us consider the ODE $3y'' + ky' - \frac{1}{y} = 0$. Since qualitative constraints have arity two or three we have to decompose equations into sub expressions. For the given ODE we get the following five equations (a) which we map to the corresponding qualitative constraints (b):

| (a) | (b) |
|---|---|
| $v_1 = y'$ | $(d/dt\ y\ v_1)$ |
| $v_2 = 3v_1'$ | $(d/dt\ v_1\ v_2)$ |
| $v_3 = kv_1$ | $(mult\ k\ v_1\ v_3)$ |
| $v_4 = \frac{1}{y}$ | $(M^-\ y\ v_4)$ |
| $v_2 + v_3 - v_4 = 0$ | $(add\ v_2\ v_3\ v_4)$ |

Because the reciprocal function $v_4$ has a negative gradient, i.e. $\frac{d}{dy}\frac{1}{y} = -\frac{1}{y^2}$, it is mapped to the $M^-$ constraint. After assembling the constraints one has to define the according landmarks for the variables $y$ and $v_1$ to $v_4$. Variables have a basic quantity space $\langle -\infty, 0, \infty \rangle$ which can be extended with user-defined landmarks. The landmarks represent domain and range boundaries of monotonic functions or denote certain constants. Furthermore, the constraints can be enhanced with corresponding values between variables. Corresponding values relate qualitative constraints with values that the constraint variables can take on simultaneously. Such corresponding values are tuples with an arity of the associated constraint. For example, the constraint on variable $v_4$ has the corresponding values $(-\infty, 0)$, $(0, \infty)$ and $(\infty, 0)$ or for an *add* constraint the triple $(a, b, c)$ states that $a + b = c$. The statement of corresponding values add further constraints and are a means to reduce the nondeterministic behavior branching during qualitative simulation. In the process of qualitative abstraction coefficients $c > 0$ of the ODE are neglected. This is because they do not influence the monotonic behavior of functions but rather act as a scale factor. For example, the coefficient 3 is ignored in function $v_2$. The choice of constants have an influence on the behavior of a QDE because they can change the sign of a term. Hence, Equation $v_3$ with constant $k$ is mapped to the according multiplication constraint. A QDE is a *structural abstraction* of an ODE if all solutions of the ODE are also solutions of the QDE. However, the reverse may not be true since different ODEs may be mapped to the same QDE.

### 3.1.3. Sign Algebra

The QSIM algorithm applies the sign algebra for solving qualitative constraints. In sign algebra the landmark 0 divides the Reals into positive (+) and negative (–) numbers. According to [112] the set of qualitative values is called the *Domain of Signs*:

$$S =_{df} \{+, 0, -\}.$$

For checking qualitative constraints against certain valuations the signs of these values with respect to certain reference points are sufficient. Therefore, we define sign-valued operators according to [112] which return the signs of values in $\mathbb{R}^*$:

- $[x]_0 = sign(x) = \begin{cases} + & \textit{if } x > 0 \\ 0 & \textit{if } x = 0 \\ - & \textit{if } x < 0. \end{cases}$
  $[x]$ is used as abbreviation for $[x]_0$ where the context is clear.
- $[x]_{x_0} = sign(x - x_0)$, where $x_0$ is a reference value for the variable $x$.
- $[\dot{x}] = [dx/dt] = sign(dx/dt)$.
- $[x]_\infty = \begin{cases} + & \textit{if } x = \infty \\ 0 & \textit{if } x \textit{ is finite} \\ - & \textit{if } x = -\infty. \end{cases}$

We can use the operators above to describe qualitative behaviors. For instance, the trace $\langle (a..b, 0), (a..b, -), (a, 0) \rangle$ of a variable $x(t)$ with landmarks $\langle 0, a, b \rangle$ can be represented as follows:

| | QSIM | $[x]_0$ | $[x]_a$ | $[x]_b$ | $[\dot{x}]$ |
|---|---|---|---|---|---|
| $x(t_0)$ | $(a..b, 0)$ | $+$ | $+$ | $-$ | $0$ |
| $x(t_0..t_1)$ | $(a..b, -)$ | $+$ | $+$ | $-$ | $-$ |
| $x(t_1)$ | $(a, 0)$ | $+$ | $0$ | $-$ | $0$ |

Here, the notation $t_0..t_1$ denotes a time interval.

Based on the domain of signs, Table 3.1 and 3.2 define the qualitative addition and multiplication respectively. While the multiplication of signs is unambiguous the addition is not. When

| add | $+$ | $0$ | $-$ |
|---|---|---|---|
| $+$ | $+$ | $+$ | $+/0/-$ |
| $0$ | $+$ | $0$ | $-$ |
| $-$ | $+/0/-$ | $-$ | $-$ |

Table 3.1.: Qualitative Addition

| mult | $+$ | $0$ | $-$ |
|---|---|---|---|
| $+$ | $+$ | $0$ | $-$ |
| $0$ | $0$ | $0$ | $0$ |
| $-$ | $-$ | $0$ | $+$ |

Table 3.2.: Qualitative Multiplication

adding opposing signs the result cannot be uniquely determined. Thus, operations in the qualitative domain are represented as relations rather than as functions. The types of the addition and multiplication are $add : S \times S \times S \mapsto Bool$ and $mult : S \times S \times S \mapsto Bool$ respectively. An elaboration of the algebraic properties of qualitative operations can be found in [112].

### 3.1.4. Behavior Inference from Qualitative Models

The whole process of qualitative simulation is mapped to a Constraint Satisfaction Problem (CSP). Given an initial state and a qualitative model (QDE), the set of next states are obtained by solving a constraint system. Then the next states from the newly discovered states are calculated and so on. The constraint system for the next state relation basically consists of two parts:

- the set of constraints from the model,
- and state transition rules imposed by the underlying theory.

The set of qualitative model constraints are an invariant for each state in the inferred behavior while the state transition rules relate the valuation of model variables between the current and the next state. For efficient constraint solving the various types of constraints are mapped to conditions over sign-algebraic properties. If a value tuple satisfies a constraint $C$, then its associated set of conditions $\{P_1,\dots,P_n\}$ must evaluate to true, i.e. $C \implies P_1 \wedge \dots \wedge P_n$. For constraint solving the reverse direction of the implication, i.e.

$$\neg P_1 \vee \dots \vee \neg P_n \implies \neg C \tag{3.8}$$

is used to filter value tuples which violate the constraint. For example, the two conditions for the monotonic function constraint $(M^+ \; x \; y)$ are:

1. $[\dot{x}] = [\dot{y}]$, provided that $x$ or $y$ is not at the endpoint of its domain. The condition states that $x$ and $y$ both have the same direction of change in the interior of the domain $[a,b]$ of the monotonic function $f$.
2. If $(x_i, y_i)$ is a pair of corresponding values, then $[x]_{x_i} = [y]_{y_i}$. This means that the two values must be on the same side of the landmarks in each pair of the corresponding values.

The condition for the qualitative derivation constraint $(d/dt \; x \; y)$ is that $[\dot{x}] = [y]_0$. The conditions states that the sign of $y$ determines the direction of change of $x$. A description of the conditions of further qualitative constraints can be found in [112].

In QSIM the domain of a QDE is defined via so called transitions. A transition consists of a condition and a transition function. The condition expresses the limits of the QDE via properties over the state variables. When a certain state satisfies the condition, the transition function is called which changes to another QDE, or when no function is defined, stops simulation. The transitions between QDEs are analogous to the mode switches in a hybrid automaton.

It has to be noted, that quantity spaces of different variables, even if they have landmarks with the same name, are completely unrelated. If there exists a relationship between landmarks of different quantity spaces it has to be expressed via explicit constraints. For instance, in order to assert that two landmarks $a$ and $b$ for $A = \langle 0, a \rangle$ and $B = \langle 0, b \rangle$ are equal one would write something like $A.a = B.b$. However, QSIM does not support to state equality between landmarks but one has to use qualitative constraints and corresponding values. For the example above the constraint $(add \; X \; Y \; Z)$ $(a,0,t),(0,b,t)$, with $X:A, Y:B, Z:$ *some quantity space with a landmark* $t$, expresses the equality between $a$ and $b$.

**Constraint Solving** In qualitative simulation new states are computed by solving Constraint Satisfaction Problems (CSPs). An introduction to CSP can be found in [142]. A CSP is a triple $(V, D, P)$, where $V = \{v_1, \ldots, v_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ are the domains of $V$, and $P$ is a set of constraints over the variables. A value tuple $(x_1, \ldots, x_n)$ is a *valid assignment* if it is consistent with all constraints in $D$. The solution of a CSP is the set of all valid assignments. CSPs can be represented as graphs. In a system with only binary constraints the vertices are the variables and the edges are the constraints. In order to express constraints with an arity larger than two, so called *hyper graphs* are used. Here, one *hyper edge* links several variables. A further possible representation is the *dual constraint graph*. In such a graph the constraints are nodes while the edges link nodes that share common variables.

A straightforward approach to solve a constraint system is to enumerate all value tuples, i.e. the crossproduct of the domains of the variables, and check if the tuples satisfy all constraints. However, because of large state spaces, this approach may even for small constraint systems be intractable. Thus, more efficient techniques like constraint propagation are applied to reduce search spaces and computation times. In constraint propagation the valid assignments to a variable $x$ are used to filter values of another variable $y$ given that $x$ and $y$ are related by some constraints. For example, $x$ and $y$ have the domains $D_x = D_y = \{0, 1, 2, 3\}$ and we have the two constraints $C_1 : x < 2, C_2 : y = x + 1$. After evaluating constraint $C_1$ we get all valid assignments to variable $x$, i.e. $\{0, 1\}$. The consistency of a variable with all unary constraints, in our example $C_1$, is called *node consistency*. The propagation of the valid assignments to $x$ from $C_1$ to $C_2$ provides *arc consistency*. Here, *arc* refers to an edge between two nodes in the constraint graph. Arc consistency ensures that two adjacent nodes assign the same value to shared variables. Hence, we get the value tuples $\{(0, 1), (1, 2)\}$ as valid assignments to the variables $(x, y)$. The extension of arc consistency is called *path consistency* where three nodes are connected by two arcs. Consistency checks can be generalized to *k-consistency* (consistency along a sequence of $k$ nodes) which, with increasing $k$, reduces backtracking during search. On the other hand, larger values for $k$ result in longer computation times for performing consistency checks which, in worst case, is exponential in $k$.

The QSIM algorithm applies constraint propagation by maintaining node and arc consistency. CSPs are represented as dual constraint graphs. According to [112] the algorithm *Cfilter* applies the following four steps:

> Given a QDE and a partial state information $\bar{D}$, generate a CSP $(V, D, P)$ and solve it. The partial state information can be provided by a user when defining an initial state or originates from successor rules during simulation.
>
> 1. (Domain restriction): Determine the domains $D_i \in D$ of each variable $v_i \in V$ by intersecting the partial state information $\bar{D}_i \in \bar{D}$ with the full set of qualitative values obtained from the quantity spaces of the variables $v_i$.
> 2. (Node consistency): The conditions $P_j$ from all QSIM constraints $C_i$ in the QDE form the set $P$. For each constraint $C_i$ with arity $k$, applying to a tuple $(v_{i_1}, \ldots, v_{i_k})$ of variables, form all tuples $(x_{i_1}, \ldots, x_{i_k})$ of values, where $x_{i_j} \in D_{i_j}$. Filter each tuple of values against each condition associated with $C_i$ according to Proposition (3.8). Tuples which violate any of the conditions are discarded.

3. (Arc consistency): For each QSIM constraint $C_i$ in the QDE, and each QSIM constraint $C_j$ sharing a common variable $v$ the following is applied: For each value tuple associated with $C_i$, where $v$ is assigned some value $x$, the tuple is discarded unless some tuple associated with $C_j$ also assigns $x$ to $v$.
4. (Exhaustive search):Generate all possible complete assignments $(x_1, \dots, x_n)$ from the remaining tuples. This gives the set of valid assignments to the CSP.

**Simulation** In order to infer the behavior of a system evolving from a given initial state, qualitative simulation is applied. A behavior is a sequence of states alternating between time points and time intervals. In general, due to nondeterminism, the result of simulating a model is a set of behaviors. Some behaviors may even not be possible to occur in any physical system. Such behaviors are called *spurious* and are artefacts resulting from qualitative abstraction. However, in our application of model-based testing this issue causes no problem since spurious behavior is filtered out during test case execution. Here, the actual observations of the implementation ensure that we cannot follow spurious behavior branches. Testing provides a constant alignment between model and reality.

The next-state-relation in qualitative behaviors is mainly governed by the continuity law, see Definition 3.4. In addition there are some global filter constraints which impose rules on sequences of states. For example, the *no change* filter states that two successive states cannot have equal valuations, see Equation 3.2. A discussion of further global filters can be found in [112]. Given the next state relation there are two possible representations of qualitative behavior:

- Behavior tree. The dynamic introduction of landmarks during simulation prevents the matching between states resulting in a tree structure. Simulation can start from one or several initial states. A tree explicitly represents the behaviors as paths from the root to the leaves.
- Envisionment. The envisionment is a graph with one or more initial states where the behaviors are paths in the graph. Furthermore, there is a distinction between attainable and total envisionment. The former starts from given initial states while the latter applies the transition relation to all possible system states.

In our work we use envisionments since they can represent infinite behavior as a finite transition system with loops. In the envisionment graph newly discovered states are matched against existing states in the graph. If there is a positive match, an according transition is added. Otherwise a transition to the new state is added to the graph.

The determination of successor states in qualitative simulation is also known as *limit analysis*. Depending on whether the current state is a time point or time interval only certain changes in the valuation of quantities can happen at the transition to a successor state [112]. Table 3.3 describes all possible state transitions from time points and Table 3.4 shows transitions possible from time intervals. The three landmarks $l_{j-1} < l_j < l_{j+1}$ are part of the quantity space of variable $v$.The successor rules ensure that the inferred behaviors comply with the continuity law. An increasing or decreasing value cannot be changed instantly. Thus, such a change will last for a time interval, which is expressed by the rules 4 to 7 in Table 3.3 for landmark and interval values respectively. The remaining rules in Table 3.3 describe the displacement of a steady value. The

rules in Table 3.4 show the possible values at the time point after a time interval. One can observe that increasing or decreasing values are unchanged, or reach an adjacent landmark, or change their sign to steady. A steady value remains unchanged.

| | $QV(v,t_i)$ | $\Longrightarrow$ | $QV(v,t_i,t_{i+1})$ |
|---|---|---|---|
| 1 | $(l_j,0)$ | | $(l_j,0)$ |
| 2 | $(l_j,0)$ | | $(l_j..l_{j+1},+)$ |
| 3 | $(l_j,0)$ | | $(l_{j-1}..l_j,-)$ |
| 4 | $(l_j,+)$ | | $(l_j..l_{j+1},+)$ |
| 5 | $(l_j,-)$ | | $(l_{j-1}..l_j,-)$ |
| 6 | $(l_j..l_{j+1},+)$ | | $(l_j..l_{j+1},+)$ |
| 7 | $(l_j..l_{j+1},-)$ | | $(l_j..l_{j+1},-)$ |
| 8 | $(l_j..l_{j+1},0)$ | | $(l_j..l_{j+1},0)$ |
| 9 | $(l_j..l_{j+1},0)$ | | $(l_j..l_{j+1},+)$ |
| 10 | $(l_j..l_{j+1},0)$ | | $(l_j..l_{j+1},-)$ |

Table 3.3.: Time Point Successors

| | $QV(v,t_i,t_{i+1})$ | $\Longrightarrow$ | $QV(v,t_{i+1})$ |
|---|---|---|---|
| 1 | $(l_j,0)$ | | $(l_j,0)$ |
| 2 | $(l_j..l_{j+1},+)$ | | $(l_{j+1},0)$ |
| 3 | $(l_j..l_{j+1},+)$ | | $(l_{j+1},+)$ |
| 4 | $(l_j..l_{j+1},+)$ | | $(l_j..l_{j+1},+)$ |
| 5 | $(l_j..l_{j+1},+)$ | | $(l_j..l_{j+1},0)$ |
| 6 | $(l_j..l_{j+1},-)$ | | $(l_j,0)$ |
| 7 | $(l_j..l_{j+1},-)$ | | $(l_j,-)$ |
| 8 | $(l_j..l_{j+1},-)$ | | $(l_j..l_{j+1},-)$ |
| 9 | $(l_j..l_{j+1},-)$ | | $(l_j..l_{j+1},0)$ |
| 10 | $(l_j..l_{j+1},0)$ | | $(l_j..l_{j+1},0)$ |

Table 3.4.: Time Interval Successors

After having a means to solve qualitative constraints and the definition of the successor state relation we present the QSIM algorithm [112]:

> Given a QDE and an initial state information, the QSIM algorithm applies the following steps for behavior inference:
>
> 1. Initialize the simulation agenda, i.e. a QSIM data structure for maintaining the current states to be explored, with the set of complete initial states consistent with the QDE and initial state information.
> 2. If the agenda is empty, or a resource limit is reached, stop the simulation. Otherwise, pop a state $S$ from the agenda.
> 3. For each variable $v_i$ in the QDE, use the qualitative successor tables to determine the possible successors to $QV(v_i,S)$. These are domain restrictions $\bar{D}_i$ on the possible qualitative values of $v_i$.
> 4. Determine all successor states $\{S_1,\ldots,S_k\}$ of $S$ consistent with the domain restrictions $\bar{D}_i$ and the constraints in the QDE. If there is a successor state $S_i$ which has the same qualitative values as $S$, then this state is removed according to the no-change rule.
> 5. Add the new states from Step 4 to the behavior graph.
> 6. Apply the global filter constraints to each successor state $S_i$. A global filter restricts the behavior between states. One example is the *No Change* filter which requires that there are no equal successive states.
> 7. Add each eligible successor state to the agenda. A state is eligible for successors if it does not satisfy any of the global filter constraints.
> 8. Continue from Step 2.

Since qualitative simulation only filters inconsistent behavior the inferred behavior is sound. After having discussed the principles of qualitative simulation we now present a small example.

**Example 3.1.** Let us consider the ODE of a harmonic oscillator $y'' + y = 0$. We can solve this equation analytically with the approach $y = e^{\lambda \cdot t}$. Applied to the ODE this yields $(\lambda^2 + 1) \cdot e^{\lambda \cdot t} = 0$. From this polynomial we get the conjugate-complex root $\lambda_{1,2} = \pm i$. Hence, the general solution of the ODE is $y(t) = c_1 \cdot e^{i \cdot t} + c_2 \cdot e^{-i \cdot t}$. By considering the boundary conditions $y(0) = 1, y'(0) = 0$ and by exploiting the connection to trigonometric functions, i.e. $e^{i \cdot t} = cos(t) + i \cdot sin(t)$, we obtain $y(t) = cos(t)$ as solution.

We now lift the example to the qualitative domain. The QSIM model in Figure 3.3 is a structural abstraction of the given ODE. Since QSIM is implemented in Lisp, the models are loaded as Lisp files. In Lines 2-5 we define three variables with quantity spaces $\langle minf, 0, inf \rangle$, where *minf* and *inf* stand for $-\infty$ and $\infty$ respectively. These course quantity spaces are sufficient for our purpose. Lines 7-10 show the qualitative constraints, where the $M^-$ constraint is associated with three corresponding value tuples. The corresponding values ensure that the values of $a$ and $c$ add up to zero according to the ODE. Thus, valuations like $a > 0$ and $c = 0$ are excluded since their sum cannot be zero. Line 12 states which variables should be displayed in the simulation output. The remaining lines specify the scenario for which the model should be simulated. The statement in Line 23 starts the qualitative simulation with a limit of 1000 states. We choose the *envisionment* mode and define an initial state where $a$ is set to a positive, decreasing value.

In a physical interpretation of the oscillator this corresponds to an initial deviation from the rest position of a system, e.g. a pendulum. The initial valuation of the remaining variables can be determined automatically by constraint solving. After loading the model file, the simulation is started by calling “(osc)” from the Lisp console. Figure 3.4 shows the behavior of the oscillator. The diagrams use the already known notation of arrows pointing up and down for increasing and decreasing behavior respectively, see Figure 3.2. Circles denote steady behavior where the first derivation is zero. The time points $T_0$ till $T_4$ partition the time axis into time intervals. The last interval after $T_4$ is the interval $T_0..T_1$. This is denoted by the ”\” in the last state of the lasso-shaped state sequence in the upper-right corner and the state number to be repeated, i.e. 3. Here, black circles are time points and white circles are time intervals. □

For this small example the qualitative behavior is a 1:1 abstraction of the continuous behavior. However, in almost all larger systems qualitative simulation will produce many behaviors in the form of transition systems. Since qualitative simulation only filters inconsistent behavior it is ensured that the solution of the continuous system will be included in the qualitative solution. On the other hand some qualitative behaviors may not describe a solution of any continuous system. As already stated before such behavior is called spurious. In the continuous domain we are interested in the solution of an ODE regarding a given state, i.e. some boundary conditions. This is called an *initial value problem*. We call the counterpart in the qualitative domain an *initial state problem*. With this notions we can formulate the concept of spurious behavior as follows: A behavior is spurious if it describes no solution of any initial value problem which is consistent with the given initial state problem.

The work in [61] deals with qualitative simulation and describes the occurrence of three kinds of spurious behaviors: spurious states, spurios transitions between states, and spurious sequences

```
1   (define-QDE oscillator
2     (quantity-spaces
3      (a (minf 0 inf))
4      (b (minf 0 inf))
5      (c (minf 0 inf)))
6
7     (constraints
8      ((d/dt a b))
9      ((d/dt b c))
10     ((M- a c) (minf inf) (0 0) (inf minf)))
11
12    (layout (a b c)))
13
14  (defun osc()
15    (with-envisioning
16    (let ((initial-state
17           (make-new-state
18            :from-qde oscillator
19            :assert-values
20            '((a ((0 inf) dec))))))
21
22      (qsim initial-state)
23      (q-continue :new-state-limit 1000)
24      (qsim-display initial-state))))
```

Figure 3.3.: QSIM model of the oscillator example.

of states. Depending on the system it is possible to design appropriate global filter constraints to prevent spurious behavior. However, it cannot be avoided in general. Say and Akin [144] have shown that qualitative simulation cannot be sound and complete. Here, completeness means that the solution of any given QDE contains no spurious behavior. The authors relate the problem of the existence of a complete QSIM algorithm to Hilbert's tenth problem:

> "Find an algorithm for deciding whether a given multivariate polynomial with integer coefficients has integer solutions".

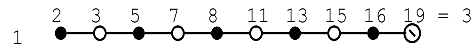
It has been proved that such an algorithm does not exist [123].

This finishes our section about qualitative simulation and we proceed with an elaboration of the algebraic properties of monotonic function constraints. We exploit these properties to simplify qualitative models.

## 3.2. Simplification of Qualitative Models

During structural abstraction from a continuous system it may be possible to simplify chains of functional relations with $n$ variables to qualitative constraints with $m$ quantities, where $m < n$. Such a chain builds a transitive link between two variables. We have developed a law that allows

Structure: cosine oscillation.
Initialization: (ONE-OF-SEVERAL-COMPLETIONS-OF S-0) (S-2)
Behavior 1 of 1: (S-2 S-3 S-5 S-7 S-8 S-11 S-13 S-15 S-16 S-19).
Final state: (GF CROSS-EDGE COMPLETE), (NIL), T<INF.



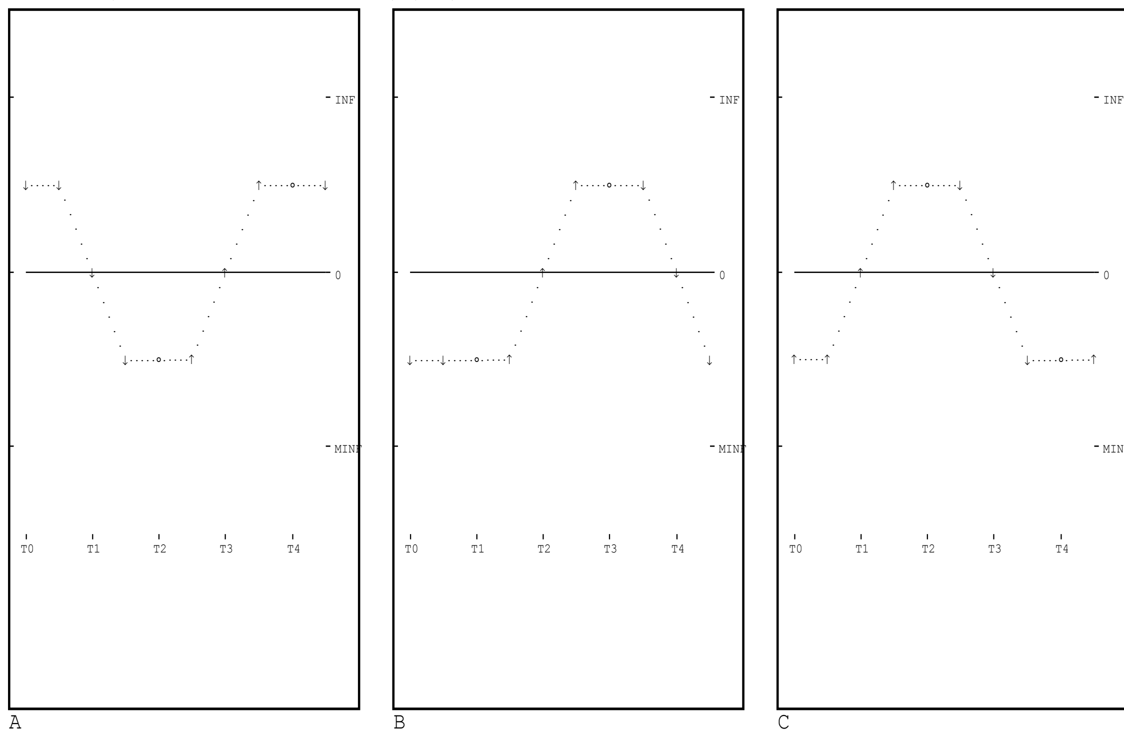




Figure 3.4.: Qualitative cosine oscillation (A), and oscillations (B) and (C) with 90° and 180° phase shift respectively.

to eliminate quantities when they occur in chained monotonic function constraints and if they are not referenced by other constraints. Let us consider the constraints $(M^+\ a\ b)$ and $(M^+\ b\ c)$. If quantity $b$ is not referenced by any other constraint we can omit $b$ and substitute both constraints by $(M^+\ a\ c)$.

In the following we use for brevity a pointfree notation. For a quantity $a$ which represents a time dependent function and a predicate $P$ we write $P(a)$ instead of $\forall t \bullet P(a(t))$. Similarly, for expressions of functions and their derivatives we may omit the argument where appropriate. For instance we write $f'$ for the first derivation of $f$ with respect to its argument which is left open.

We define the hiding of a quantity $q$ in a set of constraints $C$ as follows:

$$\textbf{hide } q \textbf{ in } C =_{df} \exists q \bullet C \tag{3.9}$$

The following definition is a generalization of the monotonic function relations, defined in Equations 3.6 and 3.7:

$$(M^s\ a\ b) =_{df} \exists f \in M \bullet b = f(a) \wedge sign(f'(a)) = s \tag{3.10}$$

Here, $s$ represents the variable for the sign and $M$ denotes the set of monotonic functions. The notations $f'(x) = \frac{d}{dx} f(x)$ stand for the derivation of $f$ with respect to its argument $x$ and if $x$ is a function we write $f' \circ x$. The following corollary expresses the resulting sign of the composition of two monotonic functions:

**Corollary 3.2** $\forall f, f_1, f_2 \in M \bullet f = f_2 \circ f_1 \implies sign(f') = sign(f_2' \circ f_1) \cdot sign(f_1')$

**Proof 3.2.** The proof follows directly from the application of the chain rule from differential calculus to function $f$:

$$\begin{aligned}
& sign((f_2 \circ f_1)') \\
\equiv \quad & \{\textit{chain rule}\} \\
& sign((f_2' \circ f_1) \cdot f_1') \\
\equiv \quad & \{\textit{sign algebra}\} \\
& sign(f_2' \circ f_1) \cdot sign(f_1')
\end{aligned}$$

□

**Lemma 3.1 (Composition of signs)**

$$f = f_2 \circ f_1 \wedge sign(f_2' \circ f_1) = x \wedge sign(f_1') = y \implies sign(f') = x \cdot y$$

**Proof 3.3.**

$$\begin{aligned}
& f = f_2 \circ f_1 \wedge sign(f_2' \circ f_1) = x \wedge sign(f_1') = y \implies sign(f') = x \cdot y \\
\equiv \quad & \\
& f = f_2 \circ f_1 \implies sign(f') = sign(f_2' \circ f_1) \cdot sign(f_1') \\
\equiv \quad & \{\textit{Corollary } 3.2\} \\
& f = f_2 \circ f_1 \implies sign(f') = sign(f') \\
\equiv \; & \textit{true}
\end{aligned}$$

□

**Theorem 3.2 (Simplification Law)** *Given two monotonic function constraints which share a common quantity. By hiding the shared quantity we obtain a new constraint which is the composition of the original ones.*

$$(\textbf{\textit{hide}}\ b\ \textbf{\textit{in}}\ (M^x\ a\ b) \wedge (M^y\ b\ c)) \implies (M^{x \cdot y}\ a\ c)$$

**Proof 3.4.**

$$
\begin{aligned}
& (\textbf{hide}\ b\ \textbf{in}\ (M^x\ a\ b) \wedge (M^y\ b\ c)) \\
\equiv \quad & \{\textit{Definition}\ (3.9), \textit{Definition}\ (3.10)\} \\
& \exists f_1, f_2 \in M, b \bullet b = f_1(a) \wedge sign(f_1'(a)) = x \wedge c = f_2(b) \wedge sign(f_2'(b)) = y \\
\equiv \quad & \{\textit{one point rule}\} \\
& \exists f_1, f_2 \in M \bullet sign(f_1'(a)) = x \wedge c = (f_2 \circ f_1)(a) \wedge sign((f_2' \circ f_1)(a)) = y \\
\equiv \quad & \{\textit{auxiliary function } f = f_2 \circ f_1\} \\
& \exists f, f_1, f_2 \in M \bullet sign(f_1'(a)) = x \wedge f = f_2 \circ f_1 \wedge c = f(a) \wedge sign((f_2' \circ f_1)(a)) = y \\
\implies \quad & \{\textit{Lemma}\ 3.1\} \\
& \exists f \in M \bullet c = f(a) \wedge sign(f'(a)) = x \cdot y \\
\equiv \quad & \{\textit{Definition}\ (3.10)\} \\
& (M^{x \cdot y}\ a\ c)
\end{aligned}
$$

□

**Theorem 3.3** *$M^+$ is an equivalence relation over the set of quantities.*

A binary relation is an equivalence relation **iff** it is reflexive, symmetric, and transitive [149], shown in the following lemmas.

**Lemma 3.2 (Reflexivity)**

$$(M^+\ a\ a)$$

**Proof 3.5.**

$$
\begin{aligned}
& (M^+\ a\ a) \\
\equiv \quad & \{\textit{Definition}\ (3.10)\} \\
& \exists f \in M \bullet a = f(a) \wedge sign(f'(a)) = + \\
\equiv \quad & \{f = id\} \\
& sign(id'(a)) = + \\
\equiv \quad & \{id'(a) = 1, sign(1) = +\} \\
& true
\end{aligned}
$$

□

In the following we denote the inverse of a function $f$ with $\bar{f}$. According to differential calculus the derivatives of inverse functions have the following property:

$$\begin{aligned} (\bar{f} \circ f)(x) = x \quad &| \frac{d}{dx} \\ (\bar{f}' \circ f)(x) \cdot f'(x) = 1 \quad &| \textit{sign algebra} \\ \textit{sign}((\bar{f}' \circ f)(x)) \cdot \textit{sign}(f'(x)) = + & \end{aligned} \tag{3.11}$$

Due to sign algebra the multiplication of equal signs results in a positive sign. Hence, we can conclude from Equation 3.11 the equality:

$$\textit{sign}((\bar{f}' \circ f)(x)) = \textit{sign}(f'(x)) \tag{3.12}$$

**Corollary 3.3 (Monotonicity of inverse functions)**

$$\exists \bar{f} \bullet a = (\bar{f} \circ f)(a) \wedge \textit{sign}((\bar{f}' \circ f)(a)) = s \iff \textit{sign}(f'(a)) = s$$

**Proof 3.6.** Corollary 3.3 follows directly from Proposition 3.12. □

**Lemma 3.3 (Symmetry)**

$$(M^+ \; a \; b) \implies (M^+ \; b \; a)$$

**Proof 3.7.**

$$\begin{aligned} & (M^+ \; a \; b) \implies (M^+ \; b \; a) \\ \equiv & \quad \{\textit{Definition } (3.10)\} \\ & \exists f_1 \bullet b = f_1(a) \wedge \textit{sign}(f_1'(a)) = + \implies \exists f_2 \bullet a = f_2(b) \wedge \textit{sign}(f_2'(b)) = + \\ \equiv & \quad \{\textit{predicate calculus}: (\exists x \bullet P(x)) \implies c \equiv \forall x \bullet P(x) \implies c\} \\ & b = f_1(a) \wedge \textit{sign}(f_1'(a)) = + \implies \exists f_2 \bullet a = f_2(b) \wedge \textit{sign}(f_2'(b)) = + \\ \equiv & \\ & \textit{sign}(f_1'(a)) = + \implies \exists f_2 \bullet a = (f_2 \circ f_1)(a) \wedge \textit{sign}((f_2' \circ f_1)(a)) = + \\ \equiv & \quad \{\textit{Corollary } 3.3\} \\ & \textit{sign}(f_1'(a)) = + \implies \textit{sign}(f_1'(a)) = + \\ \equiv & \\ & \textit{true} \end{aligned}$$

□

**Lemma 3.4 (Transitivity)**

$$(M^+ \; a \; b) \wedge (M^+ \; b \; c) \implies (M^+ \; a \; c)$$

**Proof 3.8.** This follows directly from Theorem 3.2. □

The result that $M^+$ is an equivalence relation is a required property for the structural abstraction from ODEs. In particular the equality in ODEs corresponds to the $M^+$ relation in QDEs. Furthermore, if two quantities with compatible quantity spaces are related by $M^+$ we can choose one of them as representative. This provides a further source of model simplification in addition to Theorem 3.2.

## 3.3. Modeling Continuous Systems with Garp3

The modeling and simulation tool Garp3 [44] is based on Qualitative Process Theory [71]. Garp3 is implemented in SWI Prolog and provides every means to build and simulate QR models. It has a graphical modeling language which is well suited to express cause-effect relationships without considering differential equations. However, the underlying qualitative process theory also relies on the concepts of QDEs. A detailed description of the modeling primitives can be found in the user manual [35], and [43] provides an elaborate user guide for building proper QR models. Garp3 has been used in *sustainable development* to model and analyze ecosystems. For instance, the master thesis in [21] investigates population dynamics of predator-prey models.

In Garp3 a model is comprised of a set of model fragments which are the basic blocks that describe behavior. Within a model fragment the main modeling primitives are entities, quantities, and qualitative relations. Entities are the components of the system that have certain properties expressed through associated quantities. The quantities are the qualitative variables which have associated quantity spaces. For example the entity battery has the quantities voltage, current, and charge. Model fragments are collected in a library. The idea is to reuse basic blocks which are assembled to complex systems. Model fragments have conditions and consequences. The consequences are qualitative constraints. During model simulation a fragment gets active when all its conditions are satisfied. A model fragment can be seen as a QDE where the domain is restricted by conditions. Simulation stops at states where no model fragment is active. There are three types of model fragments: static, dynamic fragments, and agent fragments. Static fragments represent the proportional relations between model quantities like that the amount of water in a vessel is proportional to the water level. They may contain all modelling primitives except so called *influences* and agents. A process describes the dynamics of a system and contains at least one influence. Influences cause changes between quantities, for example a positive flow rate into a vessel will increase the amount of liquid and hence the liquid level over time. Agents are used to model exogenous influences on the system. For instance, a quantity may increase from its minimum to its maximum or remains steady at some value. Agents can be seen as signal sources that can be connected to the model.

In Garp3 qualitative constraints are represented as unary or binary relations. The Proportionalities $P^+$ and $P^-$ correspond to the monotonic function constraints $M^+$ and $M^-$ of QSIM. Ordinal relations are expressed via *inequalities* and provide a means to constrain possible behavior. Furthermore, influences are the cause of dynamic changes in a system and provide means for integration. A positive influence $I^+(A,B)$ or negative influence $I^-(A,B)$ corresponds to the integral $B = \int A \cdot dt$ respectively $B = -\int A \cdot dt$ in the continuous domain. Here lies one difference between the qualitative constraints of Garp3 and QSIM: while Garp3 provides influences, QSIM has the time derivation constraint $d/dt$. Furthermore, in Garp3 the qualitative derivation $\delta =_{df} \{dec, std, inc\}$ stands for decreasing, steady, and increasing behavior respectively.

The initial state of a system is described with a *scenario*. Given a scenario and a model, Garp3 creates an attainable envisionment, i.e. a transition system containing all possible behaviors which may evolve over time. The user can mark states in the qualitative transition system and gets the valuation of selected quantities along the sequence of states.

In order to model the oscillator example from the previous section in Garp3 we have to rewrite

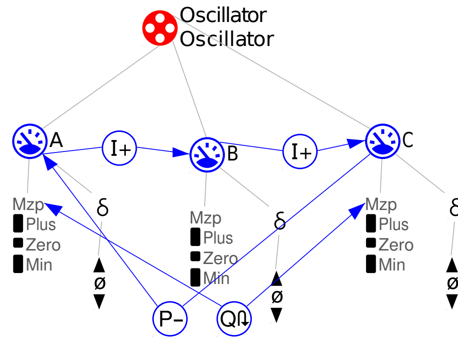



Figure 3.5.: Model of the oscillator.

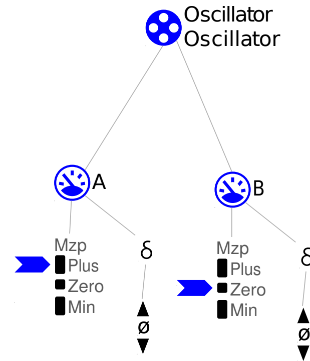



Figure 3.6.: Scenario.

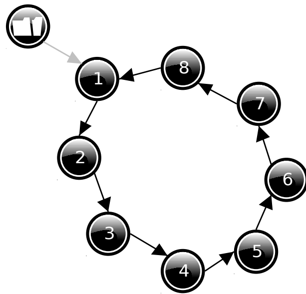



Figure 3.7.: Cycle in the transition system.

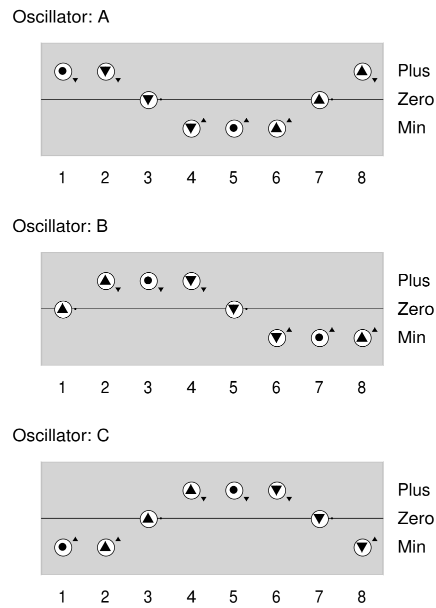



Figure 3.8.: Value history.

the given ODE into an integral equation: $y(t) + \iint y(t) \cdot dt = 0$. From this integral equation we are able to directly model the oscillator, see Figure3.5. Here we see the entity *oscillator* associated with three quantities $A \triangleq y(t)$, $B \triangleq \int y(t) \cdot dt$, and $C \triangleq \iint y(t) \cdot dt$. In contrast to QSIM, in Garp3 the intervals between landmarks have explicit names. The special landmark *Zero* denotes 0. The quantity space $\langle Min, Zero, Plus \rangle$ stands for $\langle (-\infty, 0), 0, (0, \infty) \rangle$. The two influences ($I^+$) between the quantities introduce the required integrals. The negative proportionality ($P^-$) from $C$ back to $A$ completes the equation. Similar to the QSIM oscillator model, which has corresponding values associated with the monotonic function relation, we add here an inverse value constraint. It ensures that the values of $A$ and $C$ have an inverse correspondence.

The scenario in Figure 3.6 shows the initial state. The initial value of quantity $C$ is determined by the value correspondence between $A$ and $C$. Figure 3.7 presents the state graph and Figure 3.8 shows the valuation of quantities during one cycle in the graph. In terms of control theory each integration step causes a phase shift of -90°. After twofold integration the phase shift at quantity

$C$ is -180°. Through the negative proportionality back to $A$ this is turned into a positive feedback with 0° phase shift causing the oscillation.

By rewriting derivatives to integrals we are able to model any ordinary differential equation in Garp3.

## 3.4. Qualitative Behavior - A formal Model

In the application of model-based testing we use the transition system, obtained by simulating a QR model, as formal test specification.

**Definition 3.5 (Qualitative Transition System (QTS) )** *Given a QR model, see Definition 3.1, and an initial state, then* $QTS =_{df} (S, s_0, T, v)$ *where*

- $S \subset \mathbb{N}_0$ *denotes a set of states,*
- $s_0 =_{df} 0$ *is the initial state,*
- $T : S \times S$ *is the transition relation obtained by simulating the QR model,*
- *and* $v : S \mapsto (Q \mapsto QS_q \times \delta)$ *is a valuation function binding states to value assignments for all quantities* $q \in Q$*.*

*Between two linked states exactly one quantity changes its value:*

$$\forall s_1, s_2 \in S \bullet (s_1, s_2) \in T \implies \exists! q \in Q \bullet v(s_1)(q) \neq v(s_2)(q).$$

Furthermore, we use $s \to s' =_{df} (s, s') \in T$ to denote state transitions. We define the trace semantics of a QTS as the set of all behaviors. A behavior is a sequence of qualitative values along a path of states in the QTS, starting from the initial state.

**Definition 3.6 (Trace semantics of a QTS M)**

$$traces(M) =_{df} \{\langle v(t_0), v(t_1), \ldots \rangle \mid t_i \in S \wedge t_0 = s_0 \wedge t_0 \to t_1 \to \cdots\}$$

For testing we are interested in the set of finite traces of a system. In practice, infinite traces which arise in the case of loops are bounded to a certain number of loop iterations. Based on the trace semantics we are able to define a conformance relation between two given QTS. In the following chapter we deal with the model-based testing of continuous systems.

Chapter 4

# Testing of Continuous Systems

*Parts of this chapter have been published in Brandl and Wotawa [38], Brandl et al. [40], Brandl et al. [41], Brandl et al. [39], Brandl [37], and Aichernig et al. [8].*

In the area of embedded systems, interactions with the environment are in many cases continuous. The discrete controller in a continuous environment forms a hybrid system. For example, a weather station sensing the temperature etc. or a fuel injection controller, are of this kind. When such systems are safety critical, difficult to access, or far away like Mars rovers, it is very important to test them thoroughly. Because of the system complexity of real world applications it is virtually impossible to derive enough test cases by hand in order to meet certain testing coverage metrics. We apply model-based testing and generate test cases due to test purposes or coverage criteria. The initial idea to apply qualitative models in testing of embedded systems was published in [167].

This chapter deals with the testing of continuous system environments regarding a given qualitative model. Garp3 [44] is used as modeling and simulation tool. In contrast to sequential systems, the inputs and outputs in continuous systems evolve simultaneously. It can be seen as a stream of input/output vector tuples

$$\langle([in_1,\ldots,in_n]^T,[out_1,\ldots,out_m]^T),([in_1,\ldots,in_n]^T,[out_1,\ldots,out_m]^T),\ldots\rangle.$$

We consider sampled systems where time and signal values are quantized due to certain resolutions. Furthermore, systems are strongly responsive, i.e. for every input sample exists an according output sample. Hence, in continuous systems there is no notion of quiescence like in the *ioco* theory [156] since we can always observe values on a certain output port.

## 4.1. Conformance between Qualitative Models - qrioconf

In active testing a system under test interacts with a tester, i.e. the environment, symmetrically by sending inputs and receiving outputs. The outputs of the tester are the inputs to the system and vice versa. By convention we denote outputs of the implementation as *observable* events, and

inputs to the implementation as *controllable* events. In passive testing the monitored traces of a running system are replayed on the specification. For this purpose there is no need to distinguish between input and output events since there happens no direct communication between system and tester.

However, we focus on active testing which is able to drive the SUT into certain modes which are of interest for testing. We partition the set of system quantities $Q$ into controllable quantities $L_I$, observable quantities $L_U$, and internal quantities $Q \backslash L$. Here, $L$ are the quantities at the system interface, i.e. $L =_{df} L_I \cup L_U$. For black-box testing only the visible behavior at the system interface is of relevance. We obtain the visible behavior of a qualitative transition system $M$ by projecting it onto its interface, i.e. $traces(M) \downarrow L$. For a given trace $\sigma$ over quantities $V \subseteq Q$, the operator $\alpha$ yields the set of ordered quantities of that trace, i.e. $V = \alpha(\sigma)$. In the following we refer to the values of quantities within a state in a row vector form, i.e. $[q_1, \ldots, q_n]$. For instance, the projection of the trace $\sigma = \langle [1,1],[1,2],[2,2],[2,1] \rangle \downarrow a$ with $\alpha(\sigma) = [a,b]$ yields $\langle [1],[1],[2],[2] \rangle$.

For deciding whether a continuous system conforms to a qualitative model we introduce our Qualitative Reasoning input-output conformance relation *qrioconf* [8] which is similar to the input-output conformance relation *ioconf* by Tretmans [156].

**Definition 4.1**
*Since we are interested in traces were all successive states are distinct we apply the merge function m to projected traces in order to merge equivalent states, e.g.* $m(\langle [1],[1],[2],[2] \rangle) = \langle [1],[2] \rangle$.

$$i \; \textbf{\textit{qrioconf}} \; s =_{df} \forall \sigma \in m(traces(s) \downarrow L_I) \bullet \mathbf{out}^{\sim}(i \; \mathbf{after}^{\sim} \; \sigma) \subseteq \mathbf{out}^{\sim}(s \; \mathbf{after}^{\sim} \; \sigma)$$

where $i$ is an implementation under test and $s$ is the specification (QTS). For a given input trace $\sigma \in m(traces(s) \downarrow L_I)$ the set

$$\begin{aligned} s \; \mathbf{after}^{\sim} \; \sigma =_{df} \{ t_n \mid \exists t_0, \ldots, t_n \in S, \sigma' \bullet \sigma = m(\sigma') \wedge t_0 = s_0 \wedge t_0 \to \cdots \to t_n \wedge \\ \sigma' = \langle qv_0, \ldots, qv_n \rangle \wedge qv_i = v(t_i) \downarrow \alpha(\sigma) \} \end{aligned}$$

yields all states reachable after such a trace. Here, $\mathbf{out}^{\sim}(s)$ determines the values of the observable quantities in a given state $s$, and $\mathbf{out}^{\sim}(S')$ for a given set of states $S' \subseteq S$:

$$\begin{aligned} \mathbf{out}^{\sim}(s) &=_{df} v(s) \downarrow L_U \\ \mathbf{out}^{\sim}(S') &=_{df} \{ \mathbf{out}^{\sim}(s) \mid s \in S' \} \end{aligned}$$

Figure 4.1 shows three QTS where we use integers as qualitative values. The implementation $I_1$ is qrioconf to $S$ since the outputs of $I_1$ after both input traces of $S$ are allowed, i.e.

$$\mathbf{out}^{\sim}(I_1 \; \mathbf{after}^{\sim} \; \langle [2] \rangle) = \{[1,1],[1,2]\} \subseteq \mathbf{out}^{\sim}(S \; \mathbf{after}^{\sim} \; \langle [2] \rangle) = \{[1,1],[1,2]\} \quad \text{and}$$
$$\mathbf{out}^{\sim}(I_1 \; \mathbf{after}^{\sim} \; \langle [2],[1] \rangle) = \{[1,1]\} \subseteq \mathbf{out}^{\sim}(S \; \mathbf{after}^{\sim} \; \langle [2],[1] \rangle) = \{[1,1]\}.$$

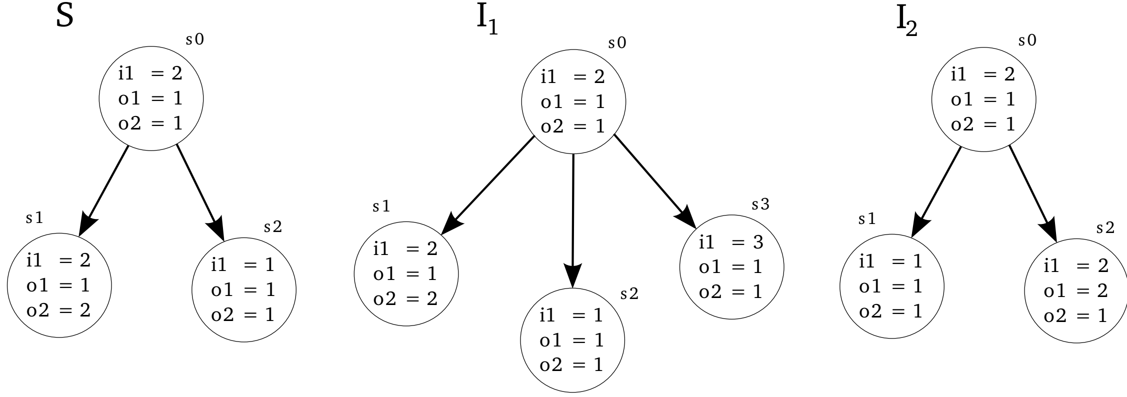



Figure 4.1.: Conformance between QR transition systems: $I_1$ qrioconf $S$ and $I_2$ ~~qrioconf~~ $S$

The input trace $\langle[2],[3]\rangle$ of $I_1$ is not specified in $S$. Due to implementation freedom the behavior following unspecified inputs is not considered by the conformance relation. For $I_2$ we get $I_2$ $\mathbf{after}$ $\langle[2]\rangle = \{s_0, s_2\}$. However, the outputs $\mathbf{out}(\{s_0, s_2\}) = \{[1,1],[2,1]\}$ of $I_2$ are not a subset of $\mathbf{out}(S\ \mathbf{after}\ \langle[2]\rangle) = \{[1,1],[1,2]\}$. Hence $I_2$ is not *qrioconf S*.

Note that a qualitative transition system $M$ is deterministic if each trace leads to at most one state, i.e. $\forall\sigma \in traces(M) \bullet |M\ \mathbf{after}\ \sigma| \leq 1$. The result of simulating a qualitative model is by definition a deterministic QTS. However, the projection of traces on the inputs leads to nondeterminism. In particular we obtain nondeterministic output behavior after a given input trace. We have developed the conformance relation *qrioconf* since an equivalence relation cannot be applied in the presence of nondeterminism.

We have implemented our qualitative testing theory in the prototype tool *QRPathfinder* which is written in *Java1.6*. The tool employs the *InterProlog*[1] interface for connecting to the QR tool *Garp3*. *QRPathfinder* loads a qualitative transition system which is obtained by simulating a *Garp3* model, generates test cases according to test purposes or coverage criteria, and provides a test adapter for executing test cases on Simulink[13] models. The details on test generation and execution are the topic of the subsequent sections.

## 4.2. Test Case Selection with Test Purposes

The simulation of qualitative models of system environments usually yields a large number of behaviors. For testing it is of interest to generate test cases for selected behaviors. This is realized via test purposes. Test purposes can be seen as a slicing criterion for extracting a part of a possibly infinite specification. For instance, the testing tool *TGV* [98] generates test cases for LOTOS [97] specifications due to test purposes. Test purpose, test case, specification, and implementation are ordered under refinement as follows:

$$TP \sqsubseteq TC \sqsubseteq Spec \sqsubseteq Impl \tag{4.1}$$

[1] http://www.declarativa.com/interprolog/

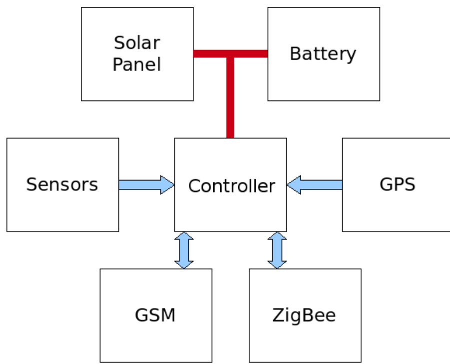



Figure 4.2.: SEPIAS Container Tracking Unit.

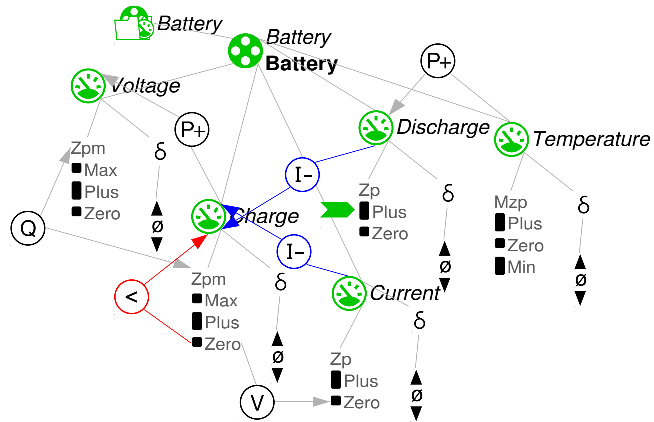



Figure 4.3.: Garp3 model of a battery.

The work in [117] discusses the relation between test purposes and test cases. A test case refines a test purpose in the context of a specification. As presented in [3] a specification can be seen as a refinement of a test case. The authors discuss the generation of mutation-based test cases in the Unified Theories of Programming (UTP) framework.

During the work in the *SEPIAS* project we have developed environmental models of a container tracking unit. The project name stands for *Self Properties In Autonomous Systems*. An excerpt of the project description says:

> The goal is to provide tools and techniques for enabling the mobile systems to react to environmental changes and internal faults in an intelligent way such that they can still fulfil their tasks. Because of limited possibilities for human intervention and even interaction during operation autonomous systems need to solve the problems and unexpected situations by their own. As a consequence, the autonomous system has to have knowledge about the surrounding environment, its task, and itself.

The block diagram in Figure 4.2 depicts the basic components of the container tracking unit. The system is powered by a solar panel and a rechargeable battery. The controller receives data from a GPS device and from some attached sensors like an acceleration sensor, a temperature sensor, etc. Based on the perception of its environment and its own state the control program adapts its operation in order to maximize the availability of its service, i.e. transmitting position data to a server. This involves power management as well as recognizing faults and according recovery by exploiting redundant services. For instance, a failed GSM connection can be replaced by routing data via near field communication (ZigBee) to another proximate container tracking unit and from there to the server.

One aspect of the continuous environment is the power supply of the system. In [40, 39] we have modeled the solar panel and the battery in Garp3. Figure 4.3 shows the model of the battery. The condition for the discharging process is satisfied if the charge level of the battery is greater than zero. A self-discharge rate, greater than zero, is defined (the arrow on the plus value) which is directly proportional to the temperature. The negative influences from the quantities *Current* and *Discharge* represent the draining of the battery. The value of *Current* is determined by some load attached to the battery. The voltage of a battery is direct proportional to its charge. There

is a value correspondence (Q arrow) between voltage and charge. Furthermore, an empty battery implies that there cannot by any load current (V arrow). This simplified model ignores the voltage drop on the internal resistance caused by a load current.

In order to generate test cases from qualitative transition systems derived from QR models, we adapt techniques from LTS testing. Therefor, we augment the transitions of a QTS with a labeling function $L : T \mapsto 2^P$, where $T$ is the transition relation and $P$ is a set of atomic propositions $S \mapsto bool$. This is similar to the definition of a Kripke structure [102], however we label transitions rather than states. The propositions are associated with symbolic names which we use to formulate test purposes in the form of regular expressions. The idea is to formally specify both test purpose and test model as LTS, and then to derive test cases by computing the synchronous product between specification and test purpose, as initially proposed by Jard and Jéron [98]. The LTS of the test purpose is obtained by converting the regular expression into its equivalent deterministic finite automaton (DFA).

We use single characters as symbolic names where 'a' $=_{df}$ *true* is reserved and stands for any proposition. Furthermore, a global proposition can be defined which is conjoined to every proposition in $P$. This is useful when specifying invariant properties. The global proposition may be referenced in the test purpose via the reserved symbol 'b'.

**Definition 4.2 (Labeling of a QTS)** *Given a QTS and a set of atomic propositions P, we augment all transitions $(s,s') \in T$ with the set of labels $\{p \in P \mid p(s')\}$. One can observe, that the label 'a' is always in this set. A transition with n labels means that we have n transitions, one for each label: $s \xrightarrow{\{a_1,\ldots,a_n\}} s' \equiv \{s \xrightarrow{a_1} s', \ldots, s \xrightarrow{a_n} s'\}$. The time complexity of the labeling algorithm equals the number of transitions times the number of propositions, i.e. $O(|T| \cdot |P|)$.*

Atomic propositions are composed of conjunctions and disjunctions of ordinal relations between quantities and their values. As an example, let us consider our battery model with a proposition stating that the charge is greater than zero, i.e. $battery : charge > zero$. Here, $battery$ is the entity name and *charge* the name of the quantity. We can also refer to the qualitative derivation of a quantity via the $dx$ operator. For instance, the proposition $battery : charge\ dx = dec$ states that the battery charge is decreasing.

The proposition symbols are used in the regular expression for specifying the test purpose. The language of the equivalent deterministic automaton are all symbol sequences that lead to an accept state. Suppose we are interested in the the cyclic occurrence of a property $p$, e.g., for three times and thereafter a path leading to property $q$. The regular expression $([\hat{}p]^*p)\{3\}[\hat{}q]^*q$ describes such a test purpose. Although theoretically possible, our current implementation does not make use of reject states, which are used, e.g. by TGV, to consider only parts of the state space during product calculation. Via reject states the product calculation between test purpose and specification can be explicitly stopped when reaching such states.

Figure 4.4 shows an example QTS consisting of three states where integers represent qualitative values. The three symbols $c$, $d$, and $e$ denote three different propositions on the state variables. The QTS is labeled according to Definition 4.2 using these symbols. Because state $s1$ only satisfies property $c$, the transition from $s0$ to $s1$ is only labeled with $c$. State $s2$ satisfies all properties, therefore the transition from $s0$ to $s2$ is labeled with all symbols. Finally, $s3$ satisfies propositions $c$ and $d$ resulting in the according labeling.

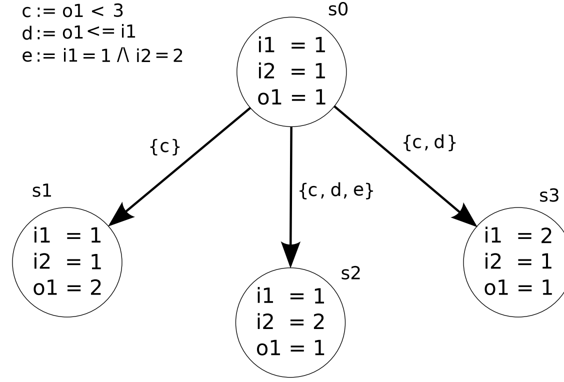



Figure 4.4.: Labeled QTS.

**Definition 4.3 (Synchronous Product)** *Let $S = (Q^S, A^S, \rightarrow_S, q_0^S)$ be a labeled QTS and $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ be a test purpose with $A^{TP} = A^S$ and $Accept^{TP} \subseteq Q^{TP}$. The synchronous product $S \times TP$ is a labeled transition system $SP = (Q^{SP}, A^{SP}, \rightarrow_{SP}, q_0^{SP})$ with $A^{SP} = A^S (= A^{TP})$, $Q^{SP} \subseteq Q^S \times Q^{TP}$, and the initial state $q_0^{SP} = (q_0^S, q_0^{TP})$. The transition relation $\rightarrow_{SP}$ is defined by:*

$$\frac{q^S \xrightarrow{a}_S q'^S \quad q^{TP} \xrightarrow{a}_{TP} q'^{TP}}{(q^S, q^{TP}) \xrightarrow{a}_{SP} (q'^S, q'^{TP})} \ .$$

*The accept states in the product are $Accept^{SP} = Q^{SP} \cap (Q^S \times Accept^{TP})$.*

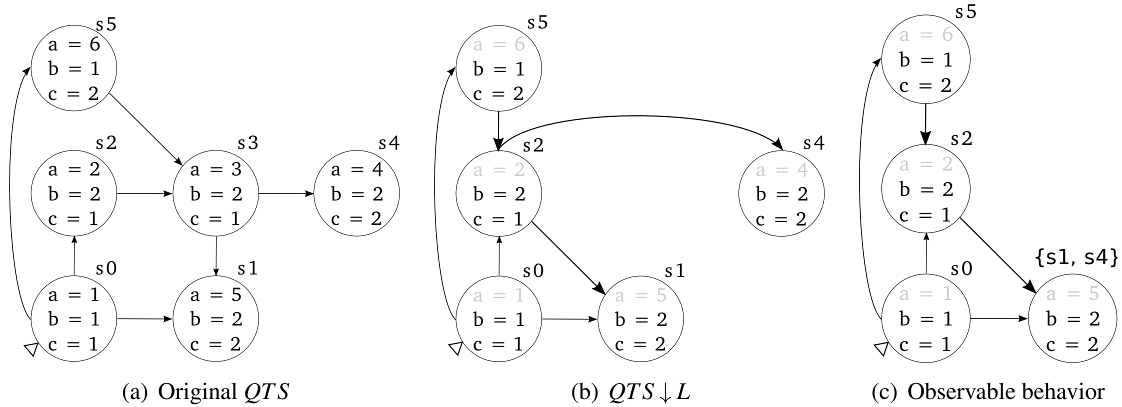
After product calculation we obtain a QTS that contains the behavior of interest for testing. We apply Tarjan's algorithm as a framework, see [104], for determining the set of states leading to an accept state. It computes the set of strongly connected components (SCCs) while updating reachability information for the visited states. A strongly connected component is defined as a subset of graph states, within which every pair of states can reach each other via transitions to states inside that set. A state can reach an accepting state if itself or another state in the same SCC can reach an accepting state. The computed subgraph is called Complete Test Graph (CTG) which consists of three types of states: states leading to an accept state, accept states, and *inconclusive* states. A state is inconclusive if there exists no path to an accept state. This means that from such a state the test purpose cannot be achieved anymore.

In the next step we project the CTG onto its visible behavior, i.e. $traces(CTG) \downarrow L$. When two connected states have the same valuation of visible quantities $L$, they are considered to be qualitatively equivalent:

$$s_1 \equiv s_2 \quad \textit{iff} \quad \forall q \in L \bullet (s_1, s_2) \in T \wedge (v.s_1).q = (v.s_2).q.$$

We merge equivalent states and update the transitions accordingly.

Figure 4.5(a) shows an example QTS with six states and three quantities. Assume that quantity $a$ is hidden. Consequently, states $s2$ and $s3$ are equivalent and can be merged. Figure 4.5(b) shows the result of this merge, with the updated transitions. Now a problem of non-determinism arises in state $s2$. The successor states $s1$ and $s4$ are equivalent and are merged due to determinization




(a) Original $QTS$ (b) $QTS \downarrow L$ (c) Observable behavior

Figure 4.5.: Minimization of a QTS.

via subset construction. The deterministic QTS representing the observable behavior is depicted in Figure 4.5(c). If all hidden quantities are constant the minimized QTS remains deterministic. This is because constant quantities cannot discriminate two states. Otherwise the QTS, in terms of visible quantities, may become non-deterministic. At the end of the state-merging process we convert the QTS into its equivalent deterministic form using the standard subset construction technique. After having a minimized, deterministic automaton, i.e. the CTG, we can extract test cases from it.

**Controllable Test Cases** According to the ioco theory test cases have to be controllable. This means that the tester never has the choice between applying an input and observing an output or between applying different inputs. Since test cases are executed synchronously with the SUT, this may cause problems. If a test case is in a state where it sends an input to the SUT and, at the same time, the implementation is sending an output this would not be allowed. However, an SUT cannot be blocked from sending outputs. In recent work [157] Jan Tretmans has given up the demand for controllability by making test cases input-complete. This means that the tester can accept all outputs of the SUT in any state. During test case selection only input controllability is ensured, i.e. in every state there is at most one input transition.

Qualitative test cases are input enabled since we treat inputs and outputs synchronously, in an alternating manner. We ensure input controllability by extracting one test case for each input decision the tester can make. Since our specifications are non-deterministic, test cases are not linear sequences but transition systems that are able to handle alternative output behavior. Since test cases may have loops the test driver has to ensure that a verdict is given after an upper bound of state transitions executed or time passed.

During test case execution a state transition occurs because of either of two reasons: the tester applies a new input value or an output quantity changes its value. Since a qualitative test case must always be able to react to changes of observed quantities they are output complete (here the outputs are inputs to the test case). If the test case has the choice between applying different inputs in the next state we derive a separate test case for each choice. In the example of Figure 4.4

**Algorithm 4.1** Qualitative Test Case

$$
\begin{aligned}
TC(s) =_{df}\ & \textbf{return}\ (s, pass) \\
& \Box \\
& Inputs := \{s' \mid \exists q \in L_I, s' \bullet (s,s') \in T \wedge (v.s).q \neq (v.s').q\}; \\
& \text{nondeterministically choose one } x \in Inputs; \\
& Next := x \cup (\{s' \mid \exists s' \bullet (s,s') \in T\} \setminus Inputs); \\
& \textbf{return}\ \{(s_1,s_2) \mid \exists s' \bullet (s_1 = s \wedge s_2 \in Next) \vee (s' \in Next \wedge (s_1,s_2) \in TC(s'))\}
\end{aligned}
$$

the valuation of input quantities $i1$ and $i2$ leads to two test cases with transitions $\{(s_0,s_1),(s_0,s_2)\}$ and $\{(s_0,s_1),(s_0,s_3)\}$ respectively.

A test purpose based test case is a qualitative transition system with two types of sink states: *pass* and *inconclusive*. The pass states are the accept states in the CTG which are reached by the test case. As already mentioned before, inconclusive states denote states from which a pass state is not reachable anymore. *Fail* states are implicit as they are reachable in every state after the observation of an unspecified event. Furthermore, a test suite $\mathcal{T}$ is a set of test cases. Algorithm 4.1 recursively defines the structure of a qualitative test case. Starting from the initial state the test case either ends in a pass state or is extended by states with possibly one input and several output value changes. By the nondeterministic choices at the selection of inputs and pass states the algorithm enumerates all possible test cases of a given QTS.

We extract test cases due to test purposes from a given CTG by applying a path search based algorithm which preserves the properties defined in Algorithm 4.1. Test case selection aims for transition coverage of the CTG.

In the soundness theorem of qualitative test cases we refer to the set of all possible traces. Given a set of quantities $L$ we define the set of all possible traces via the total envisionment of the weakest model, i.e. *true*, with $traces(L)$. The only restriction to such traces is the continuity rule, see 3.4. Remember, that the total envisionment of a qualitative model consists of all possible states connected by the transition relation. The behaviors of a total envisionment are all paths starting from any state.

The execution of a qualitative test case $t$ on an implementation $i$ is denoted with the synchronous parallel composition operator $t\rceil|i$, see [157]. An implementation fails a test case if the following holds:

$$
i \text{ fails } t =_{df} \exists\sigma \bullet t\rceil|i \stackrel{\sigma}{\Longrightarrow} fail\rceil|i'. \tag{4.2}
$$

**Theorem 4.1 (Soundness of qualitative test cases)** *A test case $t$ obtained with Algorithm 4.1 from a QTS $s$ is sound regarding* ***qrioconf*** *if*

$$
\begin{aligned}
\forall\sigma \in traces(L_I), i \bullet \exists i' \bullet (t\rceil|i \stackrel{\sigma}{\Longrightarrow} fail\rceil|i') \implies & \\
\textbf{out}^{\rightsquigarrow}(i\ \textbf{after}^{\rightsquigarrow}\ \sigma) \not\subseteq \textbf{out}^{\rightsquigarrow}(s\ \textbf{after}^{\rightsquigarrow}\ \sigma). & 
\end{aligned} \tag{4.3}
$$

**Proof 4.1.** We proof this by contradiction. Let $t$ be unsound. Then there exists an implementation $i$ such that $i$ conforms to $s$ but fails the test case:

$$
\begin{aligned}
& \exists i \bullet i \ \mathbf{qrioconf} \ s \land i \text{ fails } t \\
\equiv & \{\mathit{Definition}\ 4.2\} \\
& \exists i, i' \bullet i \ \mathbf{qrioconf} \ s \land \exists \sigma \in traces(L_I) \bullet t \rceil | i \stackrel{\sigma}{\Longrightarrow} fail \rceil | i' \\
\Longrightarrow & \{\mathit{condition\ of}\ (4.3)\} \\
& \exists i \bullet i \ \mathbf{qrioconf} \ s \land \exists \sigma \in traces(L_I) \bullet \mathbf{out}^{\approx}(i \ \mathbf{after}^{\approx} \ \sigma) \not\subseteq \mathbf{out}^{\approx}(s \ \mathbf{after}^{\approx} \ \sigma) \\
\Longrightarrow & \{\mathit{Definition}\ 4.1\} \\
& \exists i \bullet i \ \mathbf{qrioconf} \ s \land \neg i \ \mathbf{qrioconf} \ s \\
\Longrightarrow & \mathit{false}.
\end{aligned}
$$

□

According to Algorithm 4.1, during test case selection only input decisions are pruned. The set of output values in a test case after a test input trace is equal to the set of output values in the specification after the same trace, i.e. $\forall \sigma \in m(traces(t) \downarrow L_I) \bullet \mathbf{out}^{\approx}(t \ \mathbf{after}^{\approx} \ \sigma) = \mathbf{out}^{\approx}(s \ \mathbf{after}^{\approx} \ \sigma)$. This structural property preserves the soundness of generated test cases.

**Example Test Case** In order to create test cases for our battery example we have to define the inputs and outputs of the system. The battery model consists of five quantities. The charge of the battery and the discharge rate are internal quantities as they are neither directly observable nor controllable. In operation the battery current and the temperature are controlled by the battery's environment, hence they are input quantities $L_I$. We neglect the fact, that the battery temperature also depends on the load current. On the other side the battery voltage is an observable quantity $L_U$. Let us assume that the battery is charged by some fluctuating supply current. We model this by declaring the supply current as exogenous, sinusoidal changing quantity. Varying loads are emulated with a sinusoidal changing battery current. The temperature is set to some constant value. In the initial state the charge level and the load and charge currents are set to some values greater than zero.

Assume we are interested in the behavior that leads to an empty battery. Therefor, we define the two proposition symbols: $c$ for $battery : charge > zero$ and $d$ for $battery : charge = zero \land battery : charge\ dx = zero$. The regular expression $c^*d$ describes the test purpose. This test purpose is fulfilled by paths containing any number of states where the battery charge is greater than zero ($c$), followed by a state where the battery charge is zero and the battery is not charging ($d$). After simulating the model and computing the complete test graph we are ready to select test cases. Figure 4.6 shows an example test case. States are annotated with their state IDs as used in the QTS. The test case contains cyclic behavior. For test case execution this means that there must be some defined boundary after which execution stops resulting in an inconclusive verdict. Rectangle-shaped states are accept states, see state 47. The triangle-shaped state with ID 43 denotes an inconclusive state. In this state the battery is already full and is continued to be charged. This overcharging results in a sink state from which an accept state cannot be reached. Hence state 43 is marked as *inconclusive*.

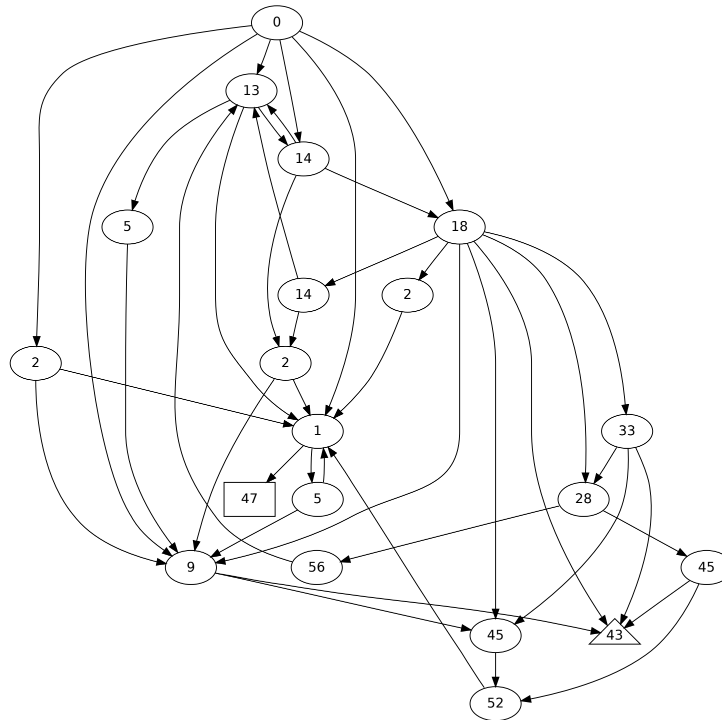



Figure 4.6.: Test case for an empty battery.

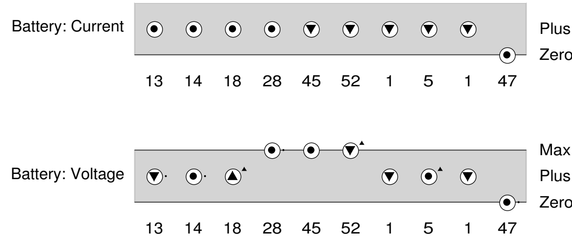



Figure 4.7.: A possible trace through the test case.

Depending on the magnitudes of the load and charge currents various paths through the test case are possible. Figure 4.7 depicts one execution sequence leading to an empty battery. The trace shows the valuation of the input and output quantities.

## 4.3. Coverage-based Test Purposes

In the field of model-based testing there always arises the question of how many test cases should be generated and what parts of the specification are covered by these test cases. The methodology discussed in the previous section requires to think of scenarios in order to formulate test purposes. For larger specifications this might be a time-consuming task and for generated test cases there is no direct relation to structural coverage on the specification.

Given a transition system derived from a QR model, we are facing the classical testing problem

of which test cases to select out of a very large possible number. In this section we deal with the automated generation of test suites according to certain coverage criteria. This allows to determine how well a system is tested with respect to its environment. The presented technique can be seen as a complementary technique for traditional testing methods.

Coverage criteria have been successfully used as stopping criteria when generating test cases, to evaluate test suites, and to automatically derive test cases. A coverage criterion describes a set of items that the testing process should exercise. Many different coverage criteria have been defined based on both source code and specifications. For example, given a transition system we might aim to cover each transition of every state at least once.

When using explicit environment models the question of which test cases to select remains. Therefore, we define a set of coverage criteria for QR models: domain coverage, delta coverage, full delta coverage, as well as the traditional state and transition coverage. Below we define a set of different coverage criteria. A coverage value according to these coverage criteria can be measured as the ratio of covered items to items in total as defined in the coverage criterion.

In a QTS, the state of the environment is given by a value assignment to the model's quantities. Each quantity $q \in Q$ has a finite domain (we omit here the abstract derivation $\delta$), its quantity space : $QS_q = \{d_0, d_1, \ldots, d_n\}$, therefore it is feasible to require each quantity to take on all of its quantity space values. This coverage criterion is called *Domain Coverage*.

**Definition 4.4 (Domain Coverage)** *A test suite $\mathcal{T}$ achieves domain coverage, if for each quantity $q \in Q$, there is a test case $t = (S^t, s_0^t, T^t, v^t) \in \mathcal{T}$ for each $d \in QS_q$, such that $\exists s \in S^t, d' : \delta \bullet v^t(s)(q) = (d, d')$.*

Because testing all possible environment states might be impractical, domain coverage offers a compromise by ensuring that each possible domain value occurs at some point. That is, given a test suite that satisfies domain coverage we know that each possible environment value has been exercised at least once.

As quantities change their values according to the description given in the QR Model, errors might only be detected when considering value changes. Consequently, we define *Delta Coverage* such that each quantity has changed its value in every direction at least once.

**Definition 4.5 (Delta Coverage)** *A test suite $\mathcal{T}$ achieves delta coverage, if for each quantity $q \in Q$, there are test cases $t_1 \in \mathcal{T}$, $t_2 \in \mathcal{T}$, and $t_3 \in \mathcal{T}$ such that $\exists d_1, d_2, d_3 : QS_q, s_1 \in S^{t_1}, s_2 \in S^{t_2}, s_3 \in S^{t_3} \bullet v^{t_1}(s_1)(q) = (d_1, dec) \wedge v^{t_2}(s_2)(q) = (d_2, std) \wedge v^{t_3}(s_3)(q) = (d_3, inc)$.*

Delta coverage ensures that every quantity eventually changes its direction. Note that in Definition 4.5 $t_1$, $t_2$, and $t_3$ can be the same test case. Changes of direction adhere to the continuity law 3.4. Sudden, non-continuous jumps on domain values or deltas cannot happen.

Delta coverage can miss cases where some value change causes an error but a different value change for the same delta is chosen for testing. Consequently, we define *Complete Delta Coverage* as a stricter variant of delta coverage.

**Definition 4.6 (Complete Delta Coverage)** *A test suite $\mathcal{T}$ achieves complete delta coverage, if for each quantity $q \in Q$ with quantity space $QS_q$, there are test cases $t_1$, $t_2$, and $t_3$ for each $d \in QS_q$ such that $\exists s_1 \in S^{t_1}, s_2 \in S^{t_2}, s_3 \in S^{t_3} \bullet v^{t_1}(s_1)(q) = (d, dec) \wedge v^{t_2}(s_2)(q) = (d, std) \wedge v^{t_3}(s_3)(q) = (d, inc)$.*

Complete delta coverage is a combination of delta and domain coverage demanding that every domain value of a quantity changes its direction. Depending on the number of quantities and the sizes of their quantity spaces, complete delta coverage might be hard to achieve. Complete delta coverage might also contain infeasible test goals; for example it might not be possible to decrease a value at the lower bound of a quantity's domain.

In addition to the above coverage criteria, we can apply traditional coverage criteria for transition systems to the QTS. *State coverage* requires that each state of the QTS is visited at least once, and *Transition coverage* requires that every transition of the QTS is executed.

**Definition 4.7 (State Coverage)** *A test suite $\mathcal{T}$ achieves state coverage for QTS $M = (S, s_0, T, v)$, if for every $s \in S$ there is a test case $t = (S^t, s_0^t, T^t, v^t) \in \mathcal{T}$ such that $s \in S^t$.*

**Definition 4.8 (Transition Coverage)** *A test suite $\mathcal{T}$ achieves transition coverage for QTS $M = (S, s_0, T, v)$, if for every $(s, s') \in T$ there is a test case $t = (S^t, s_0^t, T^t, v^t) \in \mathcal{T}$ such that $(s, s') \in T^t$.*

The presented coverage criteria can be used to evaluate existing test suites and they can also be applied to automatically derive test suites from qualitative models. For automated test case derivation we generate regular expressions specifying a test purpose which satisfies the coverage criterion. We generate test purposes and furthermore test cases for all input and output quantities of the test model. Given a set of test purposes corresponding to a coverage criterion we create test cases according to the technique, presented in the previous section.

Consider a quantity space $\langle a, \ldots, b \rangle$. With the domain's boundary values $a$ and $b$ we can formulate a test purpose $([\hat{}a]^*a[\hat{}b]^*b)|([\hat{}b]^*b[\hat{}a]^*a)$ ensuring domain coverage. It reads as follows: start from any value in the domain, try to reach one of the endpoints and from there the other end point. The disjunction in the regular expression allows to start the traversal of domain values in either direction. Such a test purpose covers all domain values of a certain quantity. If it is not possible to visit all domain values within one test case, we have to formulate test purposes as classical reachability problem targeting at single, uncovered domain values $d_i$, i.e. $[\hat{}d_i]^*d_i$.

We can describe delta coverage with the same regular expression as for domain coverage, but with fixed boundary values *dec* and *inc* since $\delta \in \{dec, std, inc\}$. In order to express complete delta coverage we generate test purposes for each possible pair of domain value and delta, e.g., $(value1, dec), (value1, std), (value1, inc), \ldots$, resulting in $\sum_{q \in Q} 3 \cdot |QS_q|$ pairs. Such a test purpose looks like $[\hat{}p]^*p$, where $p$ is the property to be searched for.

For state coverage we create test purposes like for complete delta coverage but the properties we search for are complete quantity assignments corresponding to a certain state. We generate test purposes for every state in the specification. To ensure transition coverage we use a test purpose of the form $[\hat{}p]^*pq$ for every transition in the specification. Here $p$ specifies the start state and $q$ the end state of the transition to be covered.

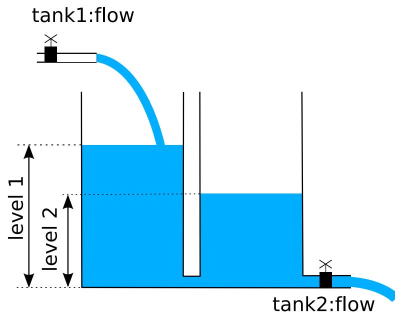



Figure 4.8.: Two-tank system.

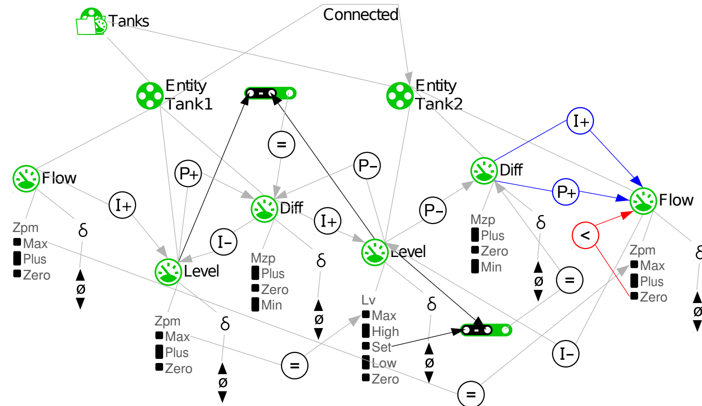



Figure 4.9.: Qualitative model for controlling the level in the tank with the outlet valve.

**Example – Two-Tank System** Consider the two-tank system from Figure 4.8. The two tanks are connected via a pipe at the bottom. Water can flow in both directions through the pipe. The flow rate depends on the difference of water levels. The control system has to hold the water level of *tank2* constant at a specified height while the water level and inflow rate of *tank1* varies. The control system can set the inflow to *tank1* and the outflow of *tank2* via controlling the valves. The water levels of both tanks are inputs of the control system. Figure 4.9 shows the Garp3 model fragment representing the physical relations and the control loop of the two-tank system.

The water level of *tank1* is the integral of the inflow rate over time, represented via a positive influence from *flow* to *level*. *Tank1* has an auxiliary quantity *diff* for calculating the difference in water levels. This difference determines the water flow through the pipe while influencing the water levels of both tanks. The quantity space of *level* in *tank2* contains a *set* point for controlling the water level. We use the auxiliary quantity *diff* of *tank2* to calculate the control deviation. The control loop is closed via a *P* and an *I* arrow to the actuator controlling the outflow valve resulting in a PI-controller. According to control theory the proportional part ensures quick response to changes and the integration part eliminates permanent control deviations. The setting of the outflow valve has a negative influence to the water level of *tank2*. Finally, equalities between the maximum tank water levels and valve flow rates makes them comparable while reducing ambiguity.

The QTS resulting from simulation of the two-tank system comprises 113 states with 305 transitions. In order to evaluate our proposed coverage criteria we use transition coverage of obtained test suites on the specification as reference measure. The results shown in Tables 4.1 and 4.2 were generated with *QRPathfinder* and list from left to right the coverage criterion, the number of generated test purposes *#TPs*, the number of obtained test purposes *#TCs*, and the transitions coverage on the specification. For Table 4.1 we exhaustively extract test cases from CTGs until all transitions of the according CTG have been considered. Table 4.2 lists each criterion for only one generated test case per test purpose.

As expected, the experiments show that the different criteria can be used to vary the amount of test cases generated from a QR model. It has to be noted that redundant test cases have not been

Table 4.1.: All Test Cases per Test Purpose

| Coverage Criterion | #TPs | #TCs | Transition Cov. | [%] |
|---|---|---|---|---|
| Domain | 5 | 120 | 220/305 | 72 |
| Delta | 5 | 133 | 217/305 | 71 |
| Complete Delta | 60 | 850 | 297/305 | 97 |
| State | 113 | 1434 | 305/305 | 100 |
| Transition | 305 | 2256 | 305/305 | 100 |

Table 4.2.: One Test Case per Test Purpose

| Coverage Criterion | #TPs | #TCs | Transition Cov. | [%] |
|---|---|---|---|---|
| Domain | 5 | 5 | 35/305 | 12 |
| Delta | 5 | 5 | 60/305 | 20 |
| Complete Delta | 60 | 42 | 119/305 | 39 |
| State | 113 | 113 | 168/305 | 55 |
| Transition | 305 | 305 | 305/305 | 100 |

filtered from the test suites. The generation of all test cases per test purpose results in quite good coverage on the model but leads also to large test suites. One can see that the large number of test cases achieve for both, state and transition coverage, the same coverage on the model, i.e. 100%. In the experiment where we generated one test case per test purpose the reached coverage values are lower. However, the test suites are significantly smaller than in Table 4.1. In the following section we deal with the execution of qualitative test cases.

## 4.4. Execution of Qualitative Test Cases

Between qualitative test cases and a continuous SUT lies big gap in abstraction. On the one hand we have no exact numerical information about the evolution of continuous variables, on the other hand qualitative behavior contains no knowledge about real time. In this section we present the design of a test driver for executing qualitative test cases.

### 4.4.1. Water Tank – A Continuous System

The water tank with two outlets, depicted in Figure 4.10, is a *switched*, continuous system. The system comprises several different modes of continuous behavior. A change between two modes is called switch. Such systems are commonly described via a set of partially defined differential equations:

$$\dot{x} = \begin{cases} i - k1\sqrt{x} - k2\sqrt{x-h} & \text{if } x \geq h \wedge v = \text{open}; \quad S0 \\ i - k2\sqrt{x-h} & \text{if } x \geq h \wedge v = \text{closed}; \quad S1 \\ i - k1\sqrt{x} & \text{if } x < h \wedge v = \text{open}; \quad S2 \\ i & \text{otherwise.} \quad S3 \end{cases} \tag{4.4}$$

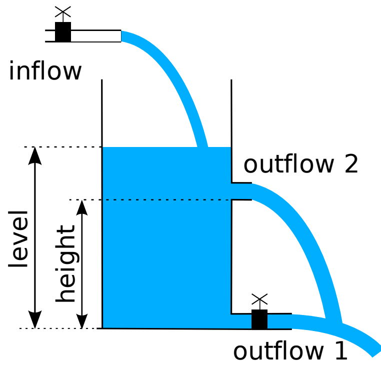



Figure 4.10.: Water Tank with two Outlets.

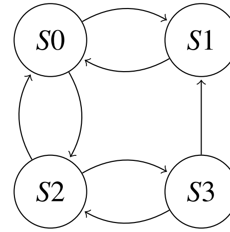



Figure 4.11.: Hybrid Automaton of the Water Tank.

The example comprises four modes $S0 - S3$, and $x$ denotes the amount of water in the tank. The outlet at the bottom can be controlled continuously with a valve $v$ between the states *open* and *closed*. The water level is the amount of water in the tank divided by the tank's base area and $h$ is the height of the upper outlet. The constants $k1$ and $k2$ are the cross sections of the two outlets. In our example we use the cross section $k1$ of the bottom outlet as variable which depends on the position of the valve.

Figure 4.11 depicts the hybrid automaton of the tank model. The labels in the four modes denote the according invariant and differential equation defined in (4.4). The edges of the automaton have no conditions. The automaton moves between states as the state invariants, i.e. the conditions associated with each partially defined differential equation, change.

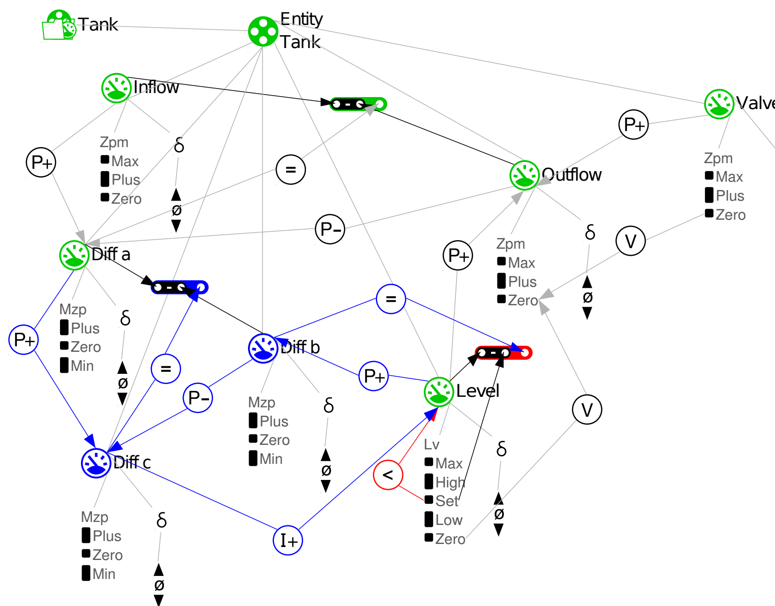



Figure 4.12.: Tank Model Fragment for Mode S0 and S1.

Next, we develop a qualitative model of the water tank. Figure 4.12 shows the Garp3 model

fragment which describes the system behavior in mode *S0* and *S1*. A second model fragment (not shown here) specifies the modes *S2* and *S3*. The effect of the valve on the outflow at the bottom is common in all four modes. Therefore, only two model fragments are needed. The further mode distinctions in the hybrid automaton result from the fact that a closed valve causes no outflow.

The entity *Tank* comprises the quantities *Inflow*, *Outflow*, *Valve*, *Level*, *Diff a*, *Diff b*, and *Diff c*. The model contains three difference relations, i.e. *Inflow* − *Outflow* = *Diff a*, *Level* − *Level.Set* = *Diff b*, and *Diff a* − *Diff b* = *Diff c*. We use them to calculate the effective flow rate *Diff c* that influences the water level. The Landmark *Level.Set* represents the level of the upper outlet. The flow rates through the outlets are direct proportional to the water level. Furthermore, correspondences between quantity space values (V-arrows) reduce ambiguities during behavior inference. For instance, there can only be an outflow at the bottom valve if the valve is not closed and there is some water in the tank.

The positive difference in height between the water level and the height of the upper outlet, i.e. *Diff b*, is proportional to a flow rate. This means that both quantities have the same monotonicity and are equivalent regarding $M^+$, see section 3.2. If in addition the values of the quantity spaces correspond to each other we can directly use *Diff b* as a flow rate. This is the case since a zero difference in height corresponds to a flow rate equal to zero, and a positive difference corresponds to a positive flow rate. The flow rate through the upper valve cannot get negative which is ensured by the condition on the water level. Furthermore, the outflow at the bottom is directly proportional to the valve position.

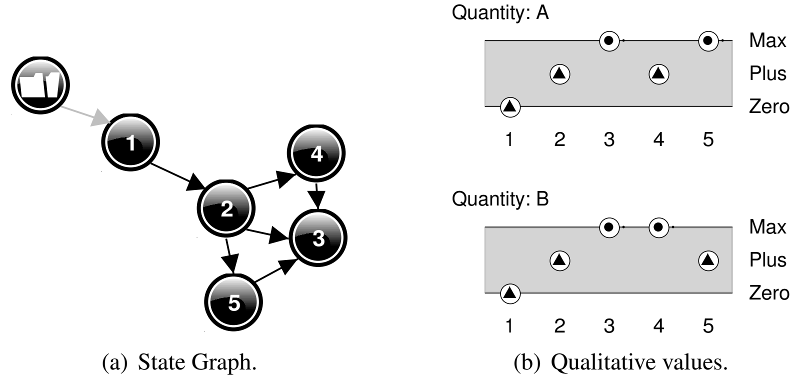
Given the qualitative model we generate test cases according to test purposes, and coverage criteria as described in Sections 4.2 and 4.3. Generated test cases have to be controllable regarding the selection of inputs.

For offline test case generation we apply the envisionment simulation, i.e. the simulation of all possible behaviors starting from an initial scenario. This can lead to quite big state spaces. In order to deal with the state space explosion problem the simulation engine can be constraint to produce fewer possible behaviors. Another option is to state additional constraints between quantities that reduce ambiguities. For instance in a two-tank-system the information that one tank is higher than the other can be expressed by a constraint between landmark values: $Tank1.Height > Tank2.Height$.

For test case execution a link has to be established between the abstract behavior of a test case and the continuous behavior of a system under test. Therefore, we have to map between the domains accordingly. Such a link has the mathematical properties of a *Galois Connection* and we describe the mapping functions in the following subsection.

### 4.4.2. Mapping between Abstract and Concrete Data

In QR the timing behavior of physical systems are represented by all possible interleavings of quantity valuations. Figure 4.13 depicts the behavior of two unrelated, increasing quantities which start from landmark *zero*. The left diagram represents the QTS and the diagram on the right side shows that valuation of all states (not to be confused with a qualitative trace). The simulation predicts three possibilities: in state sequence $\langle 1,2,5,3 \rangle$ quantity $A$ reaches the maximum before




(a) State Graph. (b) Qualitative values.

Figure 4.13.: Interleaving of two increasing Quantities.

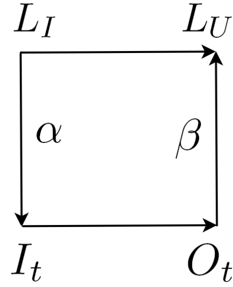



Figure 4.14.: Mapping between Abstract and Concrete Data.

quantity $B$, in state sequence $\langle 1,2,4,3 \rangle$ quantity $B$ reaches the maximum before quantity $A$, and in state sequence $\langle 1,2,3 \rangle$ both quantities increase in-phase to the maximum. Interleavings can be reduced by stating corresponding values between quantities.

Qualitative test cases are abstract in two dimensions: time and magnitude. In order to execute them on a real implementation we have to map between abstract and concrete domains. This is commonly presented in a commuting diagram, see Figure 4.14, where $\alpha$ is a refinement function and $\beta$ is an abstraction function. It states that after mapping an abstract input $i_s \in L_I$ to a concrete input $i_t \in I_t$, applying it to an implementation, and then mapping the observed output $o_t \in O_t$ to an abstract, qualitative output $o_s \in L_U$ must be the same as applying the abstract input $i_s$ to the specification. The subscript $s$ denotes state dependency and subscript $t$ stands for time dependency. In the following we denote $(o_t(t), \frac{\partial}{\partial t} o_t(t)) \in \mathbb{R} \times \mathbb{R}$ as concrete output and $i_s \in L_I = QS_{i_s} \times \delta$ as abstract (qualitative) input. For test case execution we have to associate the abstract quantity spaces of each observable quantity $q \in L_U$ with a concrete domain $CD_q$, i.e. an ordered set of real valued intervals. For instance quantity space $QS_{level}$ of a quantity *level* with $QS_{level} = \langle zero, low, medium, high, max \rangle$ corresponds to a concrete domain $CD_{level} = \langle [0,0], (0,5), [5,5], (5,10), [10,10] \rangle$. We can map between an abstract value $qVal \in QS_q$ and a real valued interval $c_i \in CD_q$ by the functions $interval_q : QS_q \mapsto CD_q$, and $quality_q : CD_q \mapsto QS_q$.

For qualitative inputs $L_I$ we refine the infinite set of monotonic increasing and decreasing func-

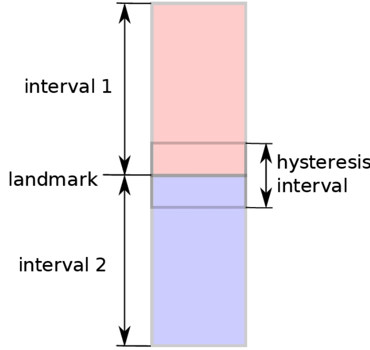



Figure 4.15.: Landmarks as Real Valued Intervals.

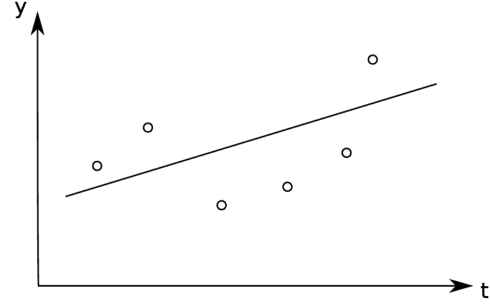



Figure 4.16.: Slope of observed Values.

tions to linear functions $I_t$ by applying the refinement function $\alpha$. For calculating the slope of $I_t$ we assume a certain time interval $TI$ within which the linear function is a refinement of the qualitative input value $(QS,\delta)$. Whenever we enter a new qualitative state we continue the piecewise linear function starting from the last concrete value of the previous state. Let us consider the following example. We want to map the abstract input $i_s = (A, plus)$ to a concrete linear function. Every abstract value matches a concrete real valued interval by definition, e.g., $A =_{df} [3.7, 5.8]$. Assume that the last concrete value $cVal = i_t(t)$ is 4.1 and the time interval is one second. Then the slope is $(5.8 - 4.1)/1 = 1.7$. By multiplying the slope with the time step *deltaT* we get the increment value *deltaY* that updates the function at each time step. The choice of the sampling interval *deltaT* must be fine-grained enough to detect the fastest signal changes of interest, see Definition 3.4.

In order to decide the output conformance of an implementation we have to abstract the observed, concrete outputs $O_t$ to qualitative outputs $L_U = QS \times \delta$ by applying the abstraction function $\beta$. Since outputs are pairs we have to deal with the two cases $\beta_1(o_t)$ and $\beta_2(\frac{\partial}{\partial t} o_t)$. We start with abstracting a concrete value $cVal = o_t(t)$ to an abstract value $qVal \in QS_q$. It is not possible to observe a constant real value from a physical system over time. Due to sensor resolution, noise, and drifts constant values will follow some distribution. In order to prevent that small oscillations of *cVals* around landmark values cause flipping between intervals, we have to blur the sharp separation due to landmark values. Therefore, we extend landmarks to intervals themselves, see Figure 4.15, and call them *hysteresis intervals*. The term *hysteresis* refers to the fact that the abstraction function is dependent on the abstract state, i.e. the last mapped value. According to the last *qVal*, denoted $qVal_{n-1}$, we can decide which new *qVal* is the right abstraction of the current concrete value. Hence, the type of the value abstraction function $\beta$ is: $\mathbb{R} \times QS \to QS$.

For example, a quantity space $\langle zero, plus, max \rangle$ can be mapped to the concrete domain $CD = [0, 3.14] \pm 0.01$ where 0.01 is the tolerance around the landmark values. The corresponding overlapping intervals are: $\langle [0, 0.02], [0.01, 3.13], [3.12, 3.14] \rangle$. Note, the special treatment of boundary values (0 and 3.14) as they cannot be expanded symmetrically: an overlap in the size of the tolerance (0.01) with their neighboring interval is generated (by setting their interval size to twice the tolerance value, here 0.02). Consequently for a quantity $q \in L_U$ with a concrete domain $CD_q$, we

can map observed *cVals* to abstract *qVals* by the following function:

$$qVal_n = \beta_1(cVal, qVal_{n-1}) = \begin{cases} qVal_{n-1} & \text{if } cVal \in c_i; \\ quality(c_j) & \text{if } \exists c_j : cVal \in c_j; \\ undef & \text{if } \neg\exists c_j : cVal \in c_j, \end{cases}$$

where $c_i = interval_q(qVal_{n-1})$ and $c_j \in CD_q \backslash \{c_i\}$.

We compute the slope of the measured quantities with a regression line of the last n samples, see Figure 4.16. The length of the delay line determines the degree of noise suppression. Depending on a defined boundary slope *bs* we abstract a gradient $\frac{\partial}{\partial t}$ to a qualitative δ. In order to improve the abstraction we use methods of curve sketching. Therefore we compute the first three derivations *dt, ddt, dddt* and map accordingly:

$$\delta_n = \beta_2(\frac{\partial}{\partial t} o_t, \delta_{n-1}) = \begin{cases} min & \text{if } dt < -bs; \\ plus & \text{if } dt > bs; \\ zero & \text{if } \neg(-bs < ddt < bs \wedge -bs < dddt < bs); \\ \delta_{n-1} & \text{otherwise.} \end{cases}$$

If *ddt* is the first non-zero derivation we can conclude the occurrence of an extremum. If *dddt* is the first non-zero derivation we have encountered a saddle point. By considering higher order derivations we get a better assurance that the first derivation is zero not only because of the choice of the boundary slope *bs*. In the case that the magnitude of all derivations is below the boundary slope we cannot conclude anything. For such points we assume the δ of the previous point ($\delta_{n-1}$).

The introduction of tolerances during abstraction is necessary because it is impossible to detect infinitesimal small changes. The tolerances have to be reflected by the system requirements, e.g. a defined boundary slope must be accurate enough to detect the smallest relevant slopes of system quantities. We consider the tolerances and the choice of the time step *deltaT* as part of the system specification. Provided that chosen values are valid assumptions, the implementation relation *qrioconf* is preserved. It can be seen analogous to the choice of the *quiescence* time interval in the *ioco* testing theory.

### 4.4.3. Test Case Execution

Within the traces of a QTS inputs and outputs are coupled. During test case execution the decision of which transition should be taken next depends on the enabledness of successor states. Either an increasing or decreasing input quantity becomes consistent with a successor state because its associates concrete function reaches a landmark value or the change of an output quantity enables the transition to a successor state.

Qualitative test cases are adaptive to implementations that differ in their timing behavior. Depending on the rise times of the signals on the implementation level different traces of a test case are executed. Note, that a possible occurrence of spurious behavior in a test case causes no problems since it is filtered out automatically during test case execution. This is because spurious behavior cannot be observed in any physical system.

Due to the nondeterministic behavior of QTS, obtained test cases are transition systems with possible loops. The implementation can decide which branch in the test case is taken next and how many iterations a loop is run through before taking a transition which exits the loop. A test case has two types of final states: *accept* and *inconclusive*. A third type, which is not represented explicitly, is the *fail* state. It can be reached from any state that is not final by completing the inputs with unexpected outputs. In an accept state the behavior of interest has been observed and from an inconclusive state the test purpose can never be satisfied.

Algorithm 4.2 describes the abstract test case execution. At first we initialize the inputs for the

**Algorithm 4.2** execute

1: $\forall q \in L_I : initialize(q.cVal)$
2: $s := initialState$
3: **while** $s \in S \wedge s \neq inconclusive \wedge s \neq accept$ **do**
4:     **for all** $q \in L_I$ **do**
5:         $(qVal, qDir) := (v.s).q$
6:         $c_i := interval_q(qVal)$
7:         **if** $qDir = zero$ **then**
8:             $deltaY := 0.0$
9:         **else if** $qDir = plus$ **then**
10:             $deltaY := c_i.max - q.cVal$
11:         **else**
12:             $deltaY := c_i.min - q.cVal$
13:         **end if**
14:         $q.deltaY := deltaY \cdot deltaT/TI$
15:     **end for**
16:     **repeat**
17:         **for all** $q \in L_I$ **do**
18:             $q.cVal := q.cVal + q.deltaY$
19:             $send(q.cVal)$
20:         **end for**
21:         **for all** $q \in L_U$ **do**
22:             $q.cVal := receive(q)$
23:         **end for**
24:     **until** $getSuccessor(s) \neq \emptyset \vee \neg(inputInvariant(s) \wedge outputInvariant(s))$
25:     $s := getSuccessor(s)$
26: **end while**
27: **if** $s = accept$ **then**
28:     **return** pass
29: **else if** $s = inconclusive$ **then**
30:     **return** inconclusive
31: **else**
32:     **return** fail
33: **end if**

implementation. To each concrete input we assign the mean value of the interval corresponding to the abstract initial value. Starting from the initial state we branch through the states of the test case until we reach a final state. For each visited state we first map the abstract inputs, i.e. (QS, $\delta$) pairs, to concrete, linear functions. This is realized in the *for loop* from line 4 to 15 by computing

the according slopes.

The *repeat-until loop* from line 16 to line 24 sends inputs to the implementation and observes outputs from the implementation. We stay in the current state until a valid successor state is encountered or the applied inputs or outputs get inconsistent with the current state. The boolean functions *inputInvariant(s)* and *outputInvariant(s)* check the consistency between abstract values and concrete trajectories. Test case execution continues until a final state is reached which is either fail, accept, or inconclusive.

Transitions between qualitative states may only be enabled for certain amounts of time. Let us consider an example test case with one input and one output quantity $\langle i, o \rangle$, both having the quantity space $\langle zero, max \rangle$ (we omit the named interval between *zero* and *max*). The test case may contain a state $s_1 = \langle (zero, std), (zero..max, dec) \rangle$ which has one successor state $s_2 = \langle (zero..max, inc), (zero..max, dec) \rangle$. If we would stay in state $s_1$ until the output quantity has decreased to zero, i.e. $o = (zero, 0)$, and becomes inconsistent with $s_1$ the window of opportunity for a state change is over. This is because there is no consistent successor state available since $\mathbf{out}^{\leadsto}(i\ \mathbf{after}^{\leadsto}\ \sigma) = (zero, 0) \nsubseteq \mathbf{out}^{\leadsto}(s\ \mathbf{after}^{\leadsto}\ \sigma) = (zero..max, dec)$ with $\sigma = \langle (zero, std) \rangle$. In such a case a fail verdict would be spurious. A fail verdict can only be issued if there exists no point in time where a successor state can be reached. Hence we traverse states in a test case by taking transitions as soon as they become enabled. This ensures that we do not miss transitions. To verify that an SUT passes a given test case it is sufficient to show that there exists one execution sequence to the test purpose. The SUT conforms to the specification if it passes all test cases from a complete test suite.

The function *getSuccessor(s)* checks the set of outgoing transitions for enabled successor states. It returns a state that is consistent with the current inputs and outputs. Depending on the reached final state the test case execution yields the verdicts:

- *pass* for an accept state,
- *fail* if execution leads to a state that is not reachable according to **qrioconf**,
- *inconclusive* for an inconclusive state.

When execution leads to an inconclusive state the verdict is given after a certain timeout interval if no successor state gets enabled. This provides us to find longer traces that may hit an accept state instead of immediately stopping with an inconclusive verdict. Furthermore, if a test case has cycles and an execution sequence does not reach a final state after some time passed or an upper bound of states visited the tester can end the execution with an inconclusive verdict.

### 4.4.4. Experimental Evaluation

We model the water tank in Matlab/Simulink[13] which we consider as our implementation under test, see Figure 4.17. In the implementation the height of the upper outlet is defined with the constant $h = 5m$. The *Gain* block represents the outflow friction of the upper outlet and is set to 3. Depending on the invariants the *selector* block switches between the partially defined differential equations accordingly. Furthermore, the model contains a scope to visualize the simulation result. We use TCP/IP blocks to communicate with our external test application. This provides a generic interface for writing different kinds of test adapters.

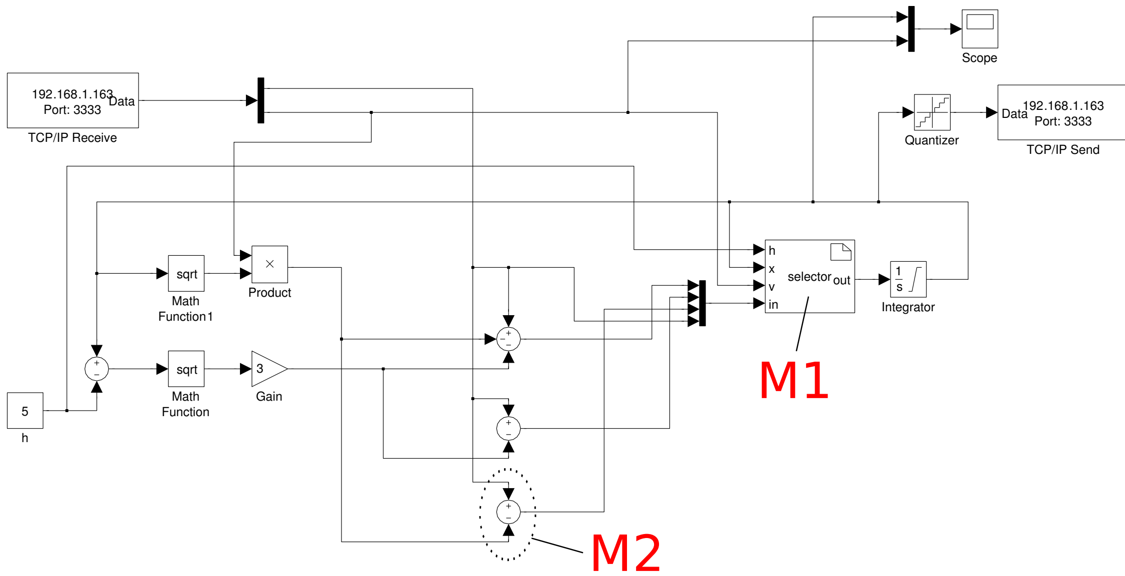


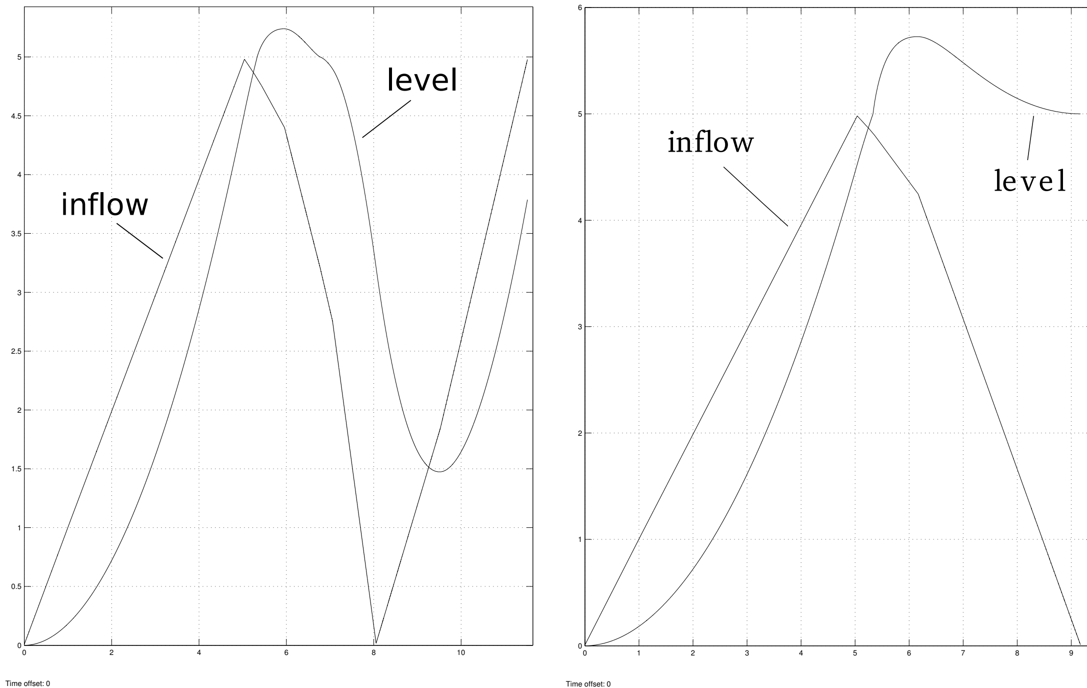

Figure 4.17.: Simulink Model of the Water Tank.

In order to evaluate the fault detection capability of our approach we create two mutants by introducing mutation *M1* and *M2* in the Simulink model, see Figure 4.17. Let us assume for mutation *M1* that a developer has swapped the invariants of modes *S0* and *S1* of the partially defined differential equations in (4.4). In mutation *M2* the plus and the minus sign in the difference node are exchanged.

We define a test scenario with the following initial values: tank level is set to *zero*, the inflow rate changes sinusoidal starting from *zero*, and the valve position is steady at *plus*. We map the input quantities *inflow* and *valve* having the quantity space $\langle zero, plus, max\rangle$ to the concrete domains $[0,5] \pm 0.01$ and $[0,3] \pm 0.01$ respectively. The output quantity *level* with $\langle zero, low, set, high, max\rangle$ corresponds to the real valued intervals $[0,5,10] \pm 0.1$.

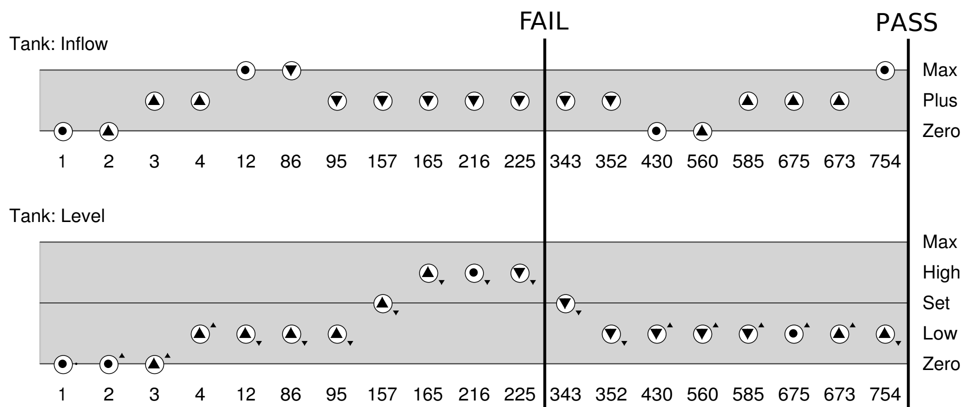
The following test purpose expresses that the inflow rate has to reach its maximum twice. Therefor we define two propositions: $c =_{df} inflow = max$ and $d =_{df} inflow \neq max$. The regular expression $(d^+c^+)\{2\}$ over the proposition symbols specifies the test purpose which accepts sequences of states containing two maxima of the inflow rate. For calculating the slope of the concrete inputs we define a time interval of one second. Furthermore, the time step *deltaT* is defined with one millisecond.

In a second scenario we set the initial tank level to a height of $8m$ which is located in the qualitative interval *high*. In addition we set the inflow rate to the steady abstract value *plus* and change the valve position sinusoidal starting from *zero*. The concrete domain for the valve position is $[0,5] \pm 0.01$. As test purpose we specify that the valve position has to reach its maximum twice. The test purpose reads similar to the one of the first test scenario. Figure 4.18 shows the input and output trajectories of the executed test case *TC1* on the implementation *I* and on the mutant *M1*. Figure 4.19 depicts the execution of *TC2* on *I* and *M2*. As can be seen both generated test cases pass the implementation and detect the mutants. Additionally, test case *TC1* kills mutant *M2* and test case *TC2* kills mutant *M1*.



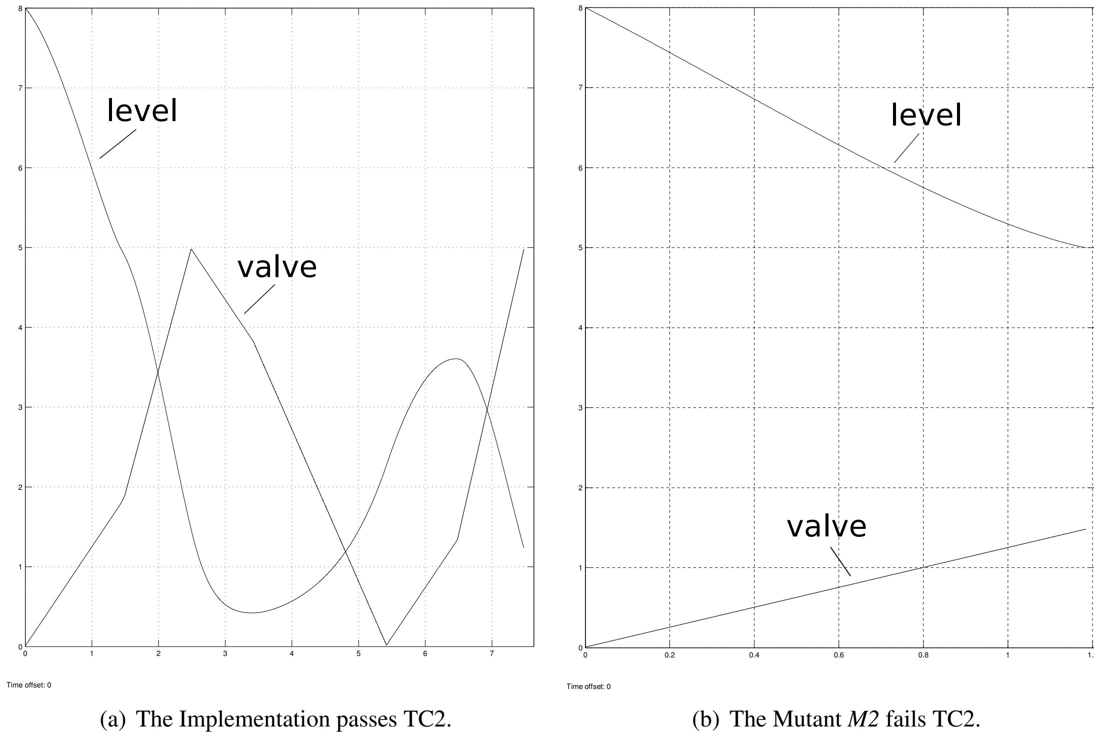

(a) The Implementation passes TC1.

(b) The Mutant *M1* fails TC1.



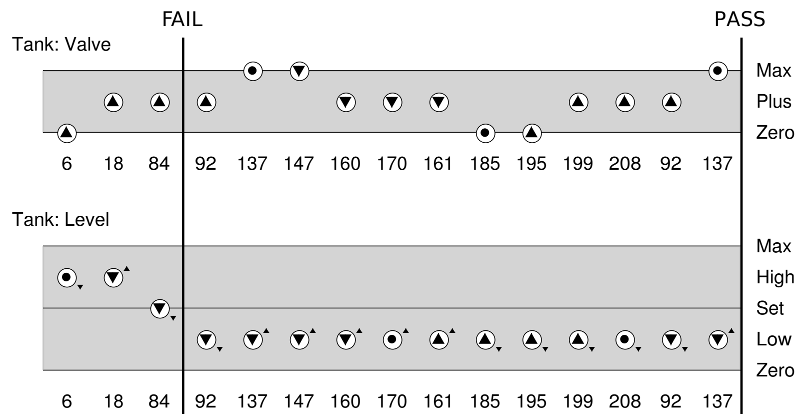

(c) Value History of TC1 executed on I and M1.

Figure 4.18.: Execution of TC1 on I and M1.




(a) The Implementation passes TC2.

(b) The Mutant *M2* fails TC2.




(c) Value History of TC2 executed on I and M2.

Figure 4.19.: Execution of TC2 on I and M2.

Chapter 5

# Testing of Hybrid Systems

*Parts of this chapter have been published in Aichernig et al. [6], Brandl et al. [42], Aichernig et al. [10], and Aichernig et al. [5].*

This chapter deals with the modeling of hybrid systems and the test case generation from such models. Hybrid systems combine both, discrete and continuous behavior. We present a new modeling approach called *Qualitative Action Systems (QAS)* which adapts *Hybrid Action Systems (HAS)* by Rönkkö et al. [139] with the technique of QR, presented in the previous sections. The underlying theory is based on the weakest precondition semantics of action systems by Back et al. [15, 16, 17].

Like other existing works in the hybrid systems area [83, 131, 139, 19] we concentrate on Ordinary Differential Equations (ODEs). Furthermore, we assume ODEs to be autonomous which means that they do not explicitly refer to the independent variable, in our case time. For instance $\frac{d}{dt}y(t) + y(t) = 0$ is an autonomous ODE while $\frac{d}{dt}y(t) + y(t) + 3t = 0$ is not. For an autonomous ODE the solution is independent of the time when the initial condition is applied. This is a required property for hybrid systems since discrete switches between ODEs may occur at any time.

The two-tank system in Figure 5.1 serves as running example through the subsequent sections. It consists of a controller operating in its continuous environment forming a hybrid system. The first step towards a system model is to know the (informal) *system requirements*:

> In the two-tank system tank $T1$ is on a lower level than the tank $T2$. $T1$ is being filled with water having some inflow rate *in*. Both tanks are connected by the pump $P1$ that is controlled such that: if the water level in $T2$ decreases below a certain *Reserve* mark and $T1$ is full, pump $P1$ starts pumping water until $T2$ is full or $T1$ gets empty. In addition, the controller needs to control the pump $P2$ that is pumping water out of $T2$: $P2$ shall be turned on as long as a button *WaterRequest* is pressed and there is enough water in $T2$ ($T2$ not $Empty$). Note, that the signal *WaterRequest* and the inflow rate *in* are not controllable by the system, hence *T1* may overflow.

Given these requirements, one is able to derive a formal model. The continuous dynamics of the

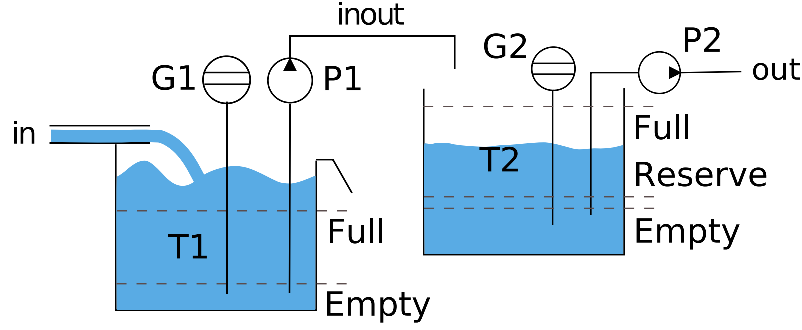



Figure 5.1.: Two-Tank Pump System.

system is expressed by the two coupled differential equations:

$$\dot{x}_1 = (in - inout)/A_1 \quad and \quad \dot{x}_2 = (inout - out)/A_2. \tag{5.1}$$

Here, $A_1$ and $A_2$ are the base areas of the two tanks and $x_1$ and $x_2$ denote the current level in the tanks. The variables *in*, *inout*, and *out* denote the flow rates into $T1$, between $T1$ and $T2$, and out of $T2$ respectively.

In the following section we present the theory of QAS and develop a QAS model of the example system.

## 5.1. Qualitative Action Systems

We start with an introduction to conventional action systems which provide a framework for describing discrete and distributed systems. The actions are statements in the form of Dijkstra's guarded commands where the semantics is defined via weakest precondition predicate transformers. An action system, see Equation 5.2, consists of a block of variable declarations followed by an initialization action $S_0$ giving to each variable an initial value, and a **do od** block looping over the nondeterministic choice of all actions. The action system *terminates* if no action is enabled. Variables declared with a star are exported by an action system and can be imported by others in the import list $I$ at the end of the action system block.

$$AS =_{df} |[\mathbf{var}\ Y : T \bullet S_0; \mathbf{do}\ A_1 \Box \ldots \Box A_n \mathbf{od}\ ]| : I \tag{5.2}$$

In order to specify distributed concurrent systems several action systems can be composed in parallel. Parallel composition of two action systems is done by joining all actions and variables. (Some variables may be shared between the systems.) As an example, the parallel composition $A^1 \parallel A^2$ of two action systems

$$A^1 = |[\mathbf{var}\ x : T^1 \bullet S_0^1; \mathbf{do}\ A_1^1 \Box \ldots \Box A_m^1 \mathbf{od}]| : u^1$$
$$A^2 = |[\mathbf{var}\ z : T^2 \bullet S_0^2; \mathbf{do}\ A_1^2 \Box \ldots \Box A_n^2 \mathbf{od}]| : u^2$$

| Action | Notation | $wp(Statement, post)$ |
|---|---|---|
| Abort | **abort** | *false* |
| Skip | **skip** | *post* |
| Assignment | $y := expr$ | $post[y := expr]$ |
| Nondet. Assignment | $y := y' \vert r$ | $\forall y' \in r.y \bullet post[y := y']$ |
| Sequential Composition | $A; B$ | $wp(A, wp(B, post))$ |
| Guarded Command | $p \rightarrow A$ | $p \implies wp(A, post)$ |
| Nondet. Choice | $A \Box B$ | $wp(A, post) \wedge wp(B, post)$ |
| Assertion | $\{p\}$ | $p \wedge post$ |
| Assumption | $[p]$ | $p \implies post$ |
| Angelic update | $\{R\}.post.\sigma$ | $\exists \gamma \in \Gamma \bullet R.\sigma.\gamma \wedge post.\gamma$ |
| Demonic update | $[R].post.\sigma$ | $\forall \gamma \in \Gamma \bullet R.\sigma.\gamma \implies post.\gamma$ |

Table 5.1.: Semantics of discrete actions.

yields $A^{1\|2}$:

$$A^{1\|2} = |[\textbf{var}\ x : T^1, z : T^2 \bullet S_0^1; S_0^2; \textbf{do}\ A_1^1 \Box \ldots \Box A_m^1 \Box A_1^2 \Box \ldots \Box A_n^2 \textbf{od}]| : (u^1 \cup u^2) \setminus (v^1 \cup v^2)$$

where $v^i$ denotes all variables exported from action system $A^i$.

Table 5.1 shows the semantics of conventional actions. Here, $\Sigma$ and $\Gamma$ are the set of states before and after a computation with predicates $p : \Sigma \mapsto Bool$ and $post : \Gamma \mapsto Bool$ respectively. The notation $term[y := x]$ denotes the textual substitution of all occurrences of $y$ in $term$ by $x$. $A, B$ and $R$ are relations between pre and post states, i.e. $\Sigma \mapsto (\Gamma \mapsto Bool)$. An action $A$ can be executed if the enabledness guard $g$ holds:

$$g(A) =_{df} \neg wp(A, false). \tag{5.3}$$

The execution of an action from states which do not satisfy the enabledness property behaves as *magic*, i.e. any postcondition may be established. When program statements $A$ are totally defined for all possible initial states they satisfy the strictness property: $wp(A, false) = false$. Furthermore, the termination guard

$$t(A) =_{df} wp(A, true) \tag{5.4}$$

ensures that when action $A$ is executed it will terminate in some post state. The guard $p$ of an action $A$ must not be weaker than its enabledness guard. This always holds when the actions are strict: $p \implies g(A) \equiv p \implies \neg wp(A, false) \stackrel{strictness}{\equiv} p \implies true \equiv true$.

### 5.1.1. Hybrid Modeling

The work in Rönkkö et al. [139] presents an approach for specifying hybrid systems within the action systems framework. The methodology is well suited for showing that one hybrid system is a refinement of another one. However, this is a (manual) proof technique. We want to check refinement between a model and an implementation by conformance testing.

Per definition, a hybrid system comprises discrete and continuous parts. In our two-tank example, the discrete part is formed by the controller that needs to start and stop different pumps. The environment, i.e. the tanks and the water, form the continuous part. We will take advantage of the compositionality of action systems to express this separation of concerns. More precisely, our hybrid model is formed as parallel composition of the controller and the environment:

$$system \equiv controller || environment.$$

This approach yields important consequences: we are working with a *closed system* and all test cases we derive are tests for the whole system.

#### Discrete Part: Controller

For a detailed controller design the requirements given in the introduction are too imprecise. We therefore need to give a more precise picture of the discrete part of our hybrid system:

1. If a button *WaterRequest* is pressed (on) and provided $T2$ is not empty (water level above *Reserve*), start pump P2 and pump water out of tank $T2$.
2. If P2 is running and *WaterRequest* is not pressed then stop P2.
3. If P2 is running and the water level of $T2$ drops to $Empty$ stop P2.
4. If tank $T2$ gets empty (water level below *Reserve* mark), and $T1$ is full then pump water out of tank $T1$ into tank $T2$ by starting pump P1.
5. If pump P1 is running and the water level in tank $T1$ drops to $Empty$ then stop P1.
6. If pump P1 is running and the water level in tank $T2$ reaches $Full$ then stop P1.

The given requirements can be mapped to a conventional action system to model the controller of the system. We need to create at least four guarded commands (actions) in order to reflect the requirements: we have to turn on/off both pumps P1 and P2. By setting a state variable *px_running* to true/false we are modeling switching on/off the pump $PX$. In addition, we set the flow rate produced by the pump $PX$. Hence, a guarded command for turning on P2 might look like

$$g_3 \rightarrow p2_running := true; out := (0, Max]$$

with $g_3$ standing for some guard and $out := (0, Max]$ for a non-deterministic assignment of some flow-rate from the interval $(0, Max]$ to the *out*-pipe. The action system of the controller can be

specified as follows.

$$
\begin{array}{llll}
Controller =_{df} |[ & \textbf{var} & & p1_running, p2_running : Bool, \\
 & & & out^*, inout^* : Real \\
 & & \bullet & p1_running := false; p2_running := false; \\
 & & & out := 0; inout := 0; wr := false \\
 & \textbf{do} & & g_1 \rightarrow p1_running := true; inout := (0, Max] \\
 & & \Box & g_2 \rightarrow p1_running := false; inout := 0 \\
 & & \Box & g_3 \rightarrow p2_running := true; out := (0, Max] \\
 & & \Box & g_4 \rightarrow p2_running := false; out := 0 \\
 & \textbf{od} & & \\
]| : x_1, x_2, wr & & &
\end{array}
$$

The two sensors for sampling the water levels and the request button for pumping water out of tank $T2$ are modeled as imported variables $x_1, x_2$ and $wr$. The given action system still has general guards $g_1$ to $g_4$ instead of concrete ones. Hence, we need to find the correct guards so that our controller behavior matches the requirements. Starting with the first requirement that specifies when P2 should be enabled we can replace $g_3$ by

$$g_3 =_{df} wr \wedge \neg p2_running \wedge x_2 > Empty$$

Requirements 2 and 3, dealing with cases when to stop P2, can be translated into guard $g_4$:

$$g_4 =_{df} p2_running \wedge (\neg wr \vee x_2 \leq Empty)$$

Similarly, $g_1$ and $g_2$ can be given as follows.

$$
\begin{aligned}
g_1 &=_{df} x_2 < Reserve \wedge x_1 = Full \wedge \neg p1_running \\
g_2 &=_{df} p1_running \wedge (x_1 < Empty \vee x_2 = Full)
\end{aligned}
$$

#### Continuous Part: Environment

While the controller can be modeled as a discrete system, the environment model depends on continuous evolutions. Hence, we have to use an extended version of conventional action systems, namely hybrid action systems, to model the environment. In HAS the continuous behavior is specified with so called *differential actions* $e :\rightarrow d$ where $e$ is the *evolution guard* and $d$ is a system of autonomous ODEs. Differential actions are atomic, i.e. the continuous state of the system evolves as long as the evolution guard is true. The evolution terminates in states where the guard does not hold anymore. Differential actions are defined via weakest precondition semantics and are relations between states before and after continuous evolutions. The states in-between are internal and hence hidden from an outside view. Combining discrete and differential actions yields a hybrid action system.

**Definition 5.1 (Hybrid Action System)**

$$HS =_{df} |[\textbf{var}\ X : T \bullet X := E; \textbf{alt}\ H\ \textbf{with}\ DH]| : I$$

*where H are discrete actions (guarded commands) and DH differential actions.*

The hybrid action system of the environment contains the differential equations describing the water flows as differential action in the **with** clause:

$$\begin{array}{llll} Environment =_{df} |[ & \textbf{var} & & x_1^*, x_2^* : Real, \\ & & \bullet & x_1 := 0; x_2 := 0; \\ & \textbf{alt} & & \\ & \textbf{with} & & true :\rightarrow \\ & & & \dot{x}_1 = (in - inout)/A_1 \wedge \dot{x}_2 = (inout - out)/A_2 \\ ]| & : & & in, out, inout \end{array}$$

Notice that in our case the evolution guard is true, hence the system will never leave the continuous evolution. However, the enabledness guard of such a differential action is *false* since enabledness requires termination. Nontermination of environmental evolutions would be a problem since we also have discrete actions in our system which would never be eligible candidates for execution. Hence, for a reactive system, it must be ensured that the evolution guard becomes false eventually. This happens when the environment changes its behavior (mode switch to another continuous action) or the controller interacts with the environment.

In order to constrain interleavings between discrete and differential actions, so called *prioritized alternation* is applied. This means that whenever a discrete action is enabled it has priority over all differential actions. For reactive systems this is an important property which allows to control the environment. Hence, the environment cannot block the controller from performing its function. In the other direction the controller eventually has to reach stable states in its computation where it interacts with the environment. Otherwise the environment and hence the progress of time is blocked. Since actions are atomic, interrupting behavior can be realized by interlocking qualitative and discrete actions. We achieve this by conjoining all negated guards of discrete actions to the evolution guards of differential actions. This execution model underlies the assumption that the controller is fast enough to reach a stable state before the next environmental interaction takes place. For details on different types of hybrid alternations, see [139, 138].

In the following we give a more formal definition of differential actions. We use this theoretical framework later on to lift differential actions to qualitative ones.

Differential actions in hybrid action systems express initial value problems of ordinary differential equations (ODEs). Given a set of continuous variables related by a set of ODEs and a set of initial values, there exists a unique solution to the initial value problem. The ODEs in differential actions are autonomous: this property means that the variable used for differentiation does not occur otherwise in the ODE. Because we are differentiating with respect to time-variable $t$, differential actions must not contain $t$.

A differential action $e :\rightarrow d$ consists of an evolution guard $e : PRED(X,Y)$ and a differential relation $d : PRED(X,\dot{X},Y)$. Both, the evolution guard and the differential relation, are predicates

over sets of discrete model variables $Y$ and continuous model variables $X$. Differential actions also provide the possibility to underspecify continuous evolutions via differential relation, e.g. $\dot{x} = [-1,1]$, meaning that $\dot{x}$ has some value in the interval $[-1,1]$. Relations over higher order derivatives can be modeled via additional variables. The following definition characterizes the evolution $\phi$ of a differential action.

**Definition 5.2 (Evolutions)**

$$\phi \textit{ is an evolution of } \quad e :\rightarrow d \quad \textit{iff} \quad SF_c(\phi, e, d) \wedge \Delta_c(\phi, e) > 0 \tag{5.5}$$

$$\begin{aligned} SF_c(\phi, e, d) =_{df} {} & \phi.0 = X \wedge \dot{\phi}.0 = \dot{X} \wedge \\ & \forall \tau : \mathbb{R}_0^+ \bullet (e \implies d)[X := \phi.\tau, \dot{X} := \dot{\phi}.\tau] \end{aligned} \tag{5.6}$$

$$\Delta_c(\phi, e) =_{df} \mathit{inf}\{\tau : \mathbb{R}_0^+ \bullet \neg e[X := \phi.\tau]\} \tag{5.7}$$

$$\textit{an evolution } \phi \textit{ is terminating} \quad \textit{iff} \quad \Delta_c(\phi, e) < \infty \tag{5.8}$$

A function $\phi$ is a solution to the differential action if it satisfies the predicate $SF_c$ as shown in Formula 5.6. The predicate demands that the function has to start at the current system state and fulfills the differential relation $d$ as long as the evolution guard is true. The termination time $\Delta_c$ (see Equation 5.7) states the boundary time when the evolution guard is not satisfied by $\phi$. Notice, that $\Delta_c$ is defined as $\Delta_c(\phi, e) =_{df} \infty$ when $\phi$ satisfies $e$ forever. Function $\phi$ is called an evolution if it satisfies Proposition 5.5 and it is said to be terminating if it lasts for a finite period of time, cf. Proposition 5.8.

The semantics of a differential action is expressed by its weakest precondition:

**Definition 5.3 (WP of Differential Actions)**

$$\begin{aligned} & wp(e :\rightarrow d, post) =_{df} \\ & \qquad \forall \phi \bullet SF_c(\phi, e, d) \wedge \Delta_c(\phi, e) > 0 \implies \\ & \qquad \Delta_c(\phi, e) < \infty \wedge post[X := \phi.(\Delta_c(\phi, e)), \dot{X} := \dot{\phi}.(\Delta_c(\phi, e))] \end{aligned}$$

The weakest precondition says that of all continuously differentiable functions, those that are evolutions for the given differential action must be terminating and end in a state fulfilling the post-condition. Because in qualitative reasoning the first derivation is part of the state space, Definition 5.3 extends the definition in [139] with $\dot{X}$ as additional state variable. The same adaptation applies to the initial value condition in (5.6), Definition 5.2.

### Putting It All Together

In order to get the complete system model, we need to compose the controller and the environment model in parallel. Unfortunately, parallel composition of hybrid action systems is more complicated than that of ordinary action systems because linear superposition of differential actions is necessary. The intuitive reason for this is the fact that only one differential action can be

executed at a time, thus the parallel composition of two systems executing two differential actions at the same time can only be modeled by superposition of the two actions.

In our two-tank example, however, we only have one differential action, hence we do not need to apply any superposition calculation. The parallel composition of *Controller* and *Environment* becomes trivial as we only need to combine the elements of both action systems, and correct the import statement. Before doing so, however, we need to fix one issue: because we want our controller (= discrete actions) to interrupt the continuous flow, we need to change the evolution guard of our differential action. Instead of making it trivially true, we need to insert the conjunction of all negated guards of the discrete actions which ensures the interrupting prioritized alternation semantics.

Merging controller and environment under interrupting prioritized alternation yields the following result. Notice, that the guards $g_1$ to $g_4$ do not change.

$$
\begin{array}{llcl}
System =_{df} |[ & \textbf{var} & & p1_running, p2_running : Bool \\
 & & & x_1, x_2, out, inout : Real \\
 & & \bullet & x_1 := 0; x_2 := 0; out := 0; inout := 0; \\
 & & & P1_running := false; P2_running := false; \\
 & \textbf{alt} & & PUMP1_ON : g_1 \rightarrow p1_running := true; inout := (0, Max] \\
 & & \Box & g_2 \rightarrow p1_running := false; inout := 0 \\
 & & \Box & g_3 \rightarrow p2_running := true; out := (0, Max] \\
 & & \Box & g_4 \rightarrow p2_running := false; out := 0 \\
 & \textbf{with} & & \neg(g_1 \vee g_2 \vee g_3 \vee g_4) :\rightarrow \\
 & & & \dot{x}_1 = (in - inout)/A_1 \wedge \dot{x}_2 = (inout - out)/A_2 \\
]| & : in, wr & &
\end{array}
$$

The hybrid model adequately represents our controlled system. The only external variable left is the inflow *in* into tank $T1$. However, for testing our requirements we are only interested in the qualitative aspects of the environment. In the following we present the qualitative abstraction of continuous behaviors in hybrid (action) systems.

#### Qualitative Action Systems

A Qualitative Action System (QAS) is obtained from a Hybrid Action System (HAS) by replacing the differential actions with *qualitative actions*.

**Definition 5.4 (Qualitative Action)** *A qualitative action $e_q :\rightharpoondown d_q$ comprises an abstract, qualitative evolution guard $e_q : PRED(Y, Q)$ and a set of qualitative differential equations $d_q : PRED(Q)$.*

Note, that in contrast to differential actions, we can not use discrete state variables $Y$ within $d_q$ because of incompatible types. Whenever the guard of a qualitative action becomes true the action is a candidate for execution by simulating (solving) the according QDEs with the current initial values, i.e. the system state. As in differential actions, the execution of a qualitative action terminates in states where the evolution guard does not hold anymore. However, in contrast to

differential actions based on differential equations, there might exist several different evolutions, all starting from the same initial value[1]. Hence, several different final states can exist.

In order to obtain the qualitative action of our example system we have to rewrite the differential equations to according QDEs. However, before we can do this, we need to think about the landmarks that describe qualitatively distinct states of the system. Fortunately, most of the landmarks are already given within Figure 5.1: the model *quantities* $x_1$ and $x_2$ have the quantity spaces $T1 = \langle 0, Empty, Full \rangle$ and $T2 = \langle 0, Empty, Reserve, Full \rangle$ respectively. For simplicity we omit an additional landmark *Overflow* for tank $T1$ and assume that when the water climbs above *Full* it will overflow. The flow rates have the quantity space $FR = \langle 0, Max \rangle$. We also need to introduce auxiliary quantities in order to be able to set up the QDEs. The auxiliary quantities are used to link different QDEs, so they only need a coarse quantity space, $NZP = \langle -\infty, 0, \infty \rangle$. It is in the response of the designer to find the right resolution of quantity spaces for expressing the system requirements appropriately.

The following qualitative constraints are written in *QSIM* like notation (predicates rather than Lisp expressions). As already discussed in Chapter 3, during qualitative abstraction constant factors, in this example the base areas of the tanks $A1$ and $A2$, are neglected. Hence, given the ordinary differential equation describing the change of the water level of the first tank in our example, $\dot{x}_1 = (in - inout)/A_1$, we can deduce following qualitative differential equations:

$$d/dt(x_1, diff_1) \land add(diff_1, inout, in)$$

The abstraction for the second ordinary differential equation is equally straight forward. By taking everything together, we get the resulting *qualitative action system.*

$$
\begin{array}{lll}
QSystem =_{df} |[ & \textbf{var} & x_1 : T1, x_2 : T2, out, inout : FR \\
 & & diff_1, diff_2 :: NZP \\
 & & P1_running, P2_running : Bool \\
 & \bullet & x_1 := (0,0); x_2 := (0,0); \\
 & & out := (0,0); inout := (0,0); \\
 & & P1_running := false; P2_running := false; \\
 & \textbf{alt} & g_1 \rightarrow P1_running := true; inout := (0..Max, 0) \\
 & \Box & g_2 \rightarrow P1_running := false; inout := (0,0) \\
 & \Box & g_3 \rightarrow P2_running := true; out := (0..Max, 0) \\
 & \Box & g_4 \rightarrow P2_running := false; out := (0,0) \\
 & \textbf{with} & \neg(g_1 \lor g_2 \lor g_3 \lor g_4) :\rightarrow \\
 & & add(diff_2, out, inout) \land add(diff_1, inout, in) \land \\
 & & d/dt(x_1, diff_1) \land d/dt(x_2, diff_2) \\
]| & : in, wr &
\end{array}
$$

Note that $x_1, x_2, out, inout$ are now quantities, i.e. qualitative model variables, and that the continuous (but qualitative) evolution of the system is interrupted every time one of the guards $g_1$ to $g_4$ becomes true.

[1] Note that differential actions based on differential *relations* may also yield several different evolutions.

After presenting the fundamental idea behind qualitative action systems, we discuss the exact semantics in the remainder of this subsection. Similar to differential actions, solutions to qualitative actions have the following properties:

**Definition 5.5 (Qualitative Evolutions)** *Let $\sigma$ be a qualitative trace and $\sigma[i]$ the valuation of the i-th state in $\sigma$.*

$$\sigma \textit{ is an evolution of } e_q :\neg\!\!\to d_q \quad \textit{iff} \quad SF_q(\sigma, e_q, d_q) \wedge \Delta_q(\sigma, e_q) > 0 \tag{5.9}$$

$$\textit{an evolution } \sigma \textit{ is terminating} \quad \textit{iff} \quad \Delta_q(\sigma, e_q) < \infty \tag{5.10}$$

$$SF_q(\sigma, e_q, d_q) =_{df} \sigma[0] = Q \wedge \forall s : dom(\sigma) \bullet (e_q \implies d_q)[Q := \sigma[s]] \tag{5.11}$$

$$\Delta_q(\sigma, e_q) =_{df} \begin{cases} i & i \in \mathbb{N}_0 \wedge \neg e_q[Q := \sigma[i]] \wedge \forall 0 \leq j < i \bullet e_q[Q := \sigma[j]] \\ 0 & else \end{cases} \tag{5.12}$$

Predicate $SF_q$ (5.11) states that a qualitative trace $\sigma$ is a valid solution to the qualitative action if it is contained in the solution of the qualitative differential equation whenever the evolution guard is satisfied. Furthermore the initial value must match the current system state. Formula (5.12) takes a qualitative trace and returns the state number when it first violates the evolution guard. Propositions (5.9) and (5.10) define when a qualitative trace is an evolution and terminating respectively.

Similar to differential actions, we define the weakest precondition of a qualitative action as follows:

**Definition 5.6 (WP of Qualitative Actions)** *Let $\psi$ be a qualitative transition system obtained by simulating $d_q$ from a given initial state.*

$$\begin{aligned} & wp(e_q :\neg\!\!\to d_q, post) =_{df} \\ & \qquad \forall \psi : QTS \bullet \forall \sigma \in traces(\psi) \bullet SF_q(\sigma, e_q, d_q) \wedge \Delta_q(\sigma, e_q) > 0 \implies \\ & \qquad \Delta_q(\sigma, e_q) < \infty \wedge post[Q := \sigma.\Delta_q(\sigma, e_q)] \end{aligned}$$

The definition specifies all initial states from which all evolutions of the given qualitative action terminate in states satisfying the post condition *post*.

The execution of a qualitative action $e_q :\neg\!\!\to d_q$ can be seen as a nondeterministic assignment $Q := \sigma[i]$ where $\sigma \in traces(M)$, $M$ is the simulated behavior of $d_q$, and $i = \Delta(\sigma, e_q)$. It states that the qualitative state $Q$ is updated to a new state $Q'$ which is the termination state of $\sigma$. If there exists no such state $Q'$, then the action does not terminate.

As already mentioned the parallel composition of differential/qualitative actions has to be treated differently than for discrete actions. While discrete actions are combined via nondeterministic choice, *qualitative composition* is applied to qualitative actions. The composition of two qualitative actions is defined as follows:

**Definition 5.7** *Let $qual_1 = e_1 :\rightharpoondown d_1$ and $qual_2 = e_2 :\rightharpoondown d_2$ be two qualitative actions then their composition, denoted by $qual_1 \boxplus qual_2$, is defined as follows*

$$qual_1 \boxplus qual_2 =_{df} \left( \begin{array}{ll} & (e_1 \wedge \neg e_2) :\rightharpoondown d_1 \\ \Box & (e_1 \wedge e_2) :\rightharpoondown (d_1 \wedge d_2) \\ \Box & (\neg e_1 \wedge e_2) :\rightharpoondown d_2 \end{array} \right)$$

The result is a nondeterministic choice over three actions with disjoint guards: Either the first action is enabled, both actions are enabled, or the second action is enabled. In the case where both actions are enabled the corresponding QDEs are conjoined. It has to be noted, that a composite QDE can be inconsistent although the individual QDEs are consistent. Here, consistent means that the qualitative constraint system has a solution. Since the enabledness guard of an inconsistent QDE does not hold the according action is never executed.

### 5.1.2. Refinement of Qualitative Actions

For blackbox testing the behavior of a system might be specified on different abstraction levels. This abstraction level determines the data refinement relation which is usually implemented in the test adapter. This means that observed events from the implementation are translated to events in the specification language and vice versa. In the following we describe the refinement between qualitative and differential actions in more detail.

The work in [63] studies simulation-based proof techniques and their implications on soundness and completeness of data refinement. It has been shown that $L$ simulation $\alpha^{-1};C \subseteq A;\alpha^{-1}$ (also called *downward* or *forward* simulation) and $L^{-1}$ simulation $C;\alpha \subseteq \alpha;A$ (also called *upward* or *backward* simulation) are sound and jointly complete. Here, $\alpha$ is an abstraction relation, $A$ is an abstract, and $C$ a concrete action. The semicolon denotes sequential composition between relations. For model-based testing we apply $U$ simulation $\alpha^{-1};C;\alpha \subseteq A$ which means that the current abstract state is refined, the concrete action $C$ is executed, and the resulting state is abstracted to the specification level. This state must be included in the set of states reachable after executing the corresponding abstract action $A$. The principle of U simulation is shown in Figure 4.14.

When the abstraction relation $\alpha$ is functional, which is the case for $v_abs_q$ (see Definition 3.2), $L$ simulation implies $U$ simulation, see [63]. Data refinement is shown by applying the following data refinement relation $r$:

**Definition 5.8 (Refinement Relation)**

$$r =_{df} Q = \alpha.(X, \dot{X}) \quad where \quad \alpha.(\phi.t, \dot{\phi}.t) =_{df} (v_abs_q.\phi).t$$

As described in [143] the weakest precondition semantics of differential actions covers not only the relation between pre- and post-states but also the flow between these states. This provides an ordering on the pre-states with respect to time. However, since the points of observation are only at pre/post states on the action level the intermediate flow states are hidden.

In order to characterize refinement between qualitative and hybrid action systems it is worthwhile to note that both can be rewritten into a pre/post condition normal form:

**Lemma 5.1 (Conjunctive Normal Form)** *Since both, differential actions C and qualitative actions A are conjunctive predicate transformers they can be rewritten into a normal form [18]: $\{p\};[R]$. Here, the predicate p is an assert statement establishing the precondition. If the precondition holds the demonic update statement $[R]$ is executed and the statement aborts otherwise.*

$$
\begin{aligned}
& C = \{p_C\};[R_C] = \\
& \{\forall \phi : C^1 \bullet SF_c(\phi, e, d) \wedge \Delta_c(\phi, e) > 0 \Rightarrow \Delta_c(\phi, e) < \infty\}; \\
& \exists \phi : C^1 \bullet SF_c(\phi, e, d) \wedge \Delta_c(\phi, e) > 0 \wedge X := \phi.(\Delta_c(\phi, e)) \wedge \dot{X} := \dot{\phi}.(\Delta_c(\phi, e))
\end{aligned}
$$

*and*

$$
\begin{aligned}
& A = \{p_A\};[R_A] = \\
& \{\forall \psi : QTS \bullet \forall g \in traces.\psi \bullet SF_q(g, e_q, d_q) \wedge \Delta_q(g, e_q) > 0 \Rightarrow \Delta_q(g, e_q) < \infty\}; \\
& \exists \psi : QTS, g \in traces.\psi \bullet SF_q(g, e_q, d_q) \wedge \Delta_q(g, e_q) > 0 \wedge Q' = g.\Delta_q(g, e_q)
\end{aligned}
$$

*The rewriting of wp semantics into normal form is straightforward and can be found in [139].*

Given this normal form, the following refinement law expresses the well-known fact that under refinement preconditions are weakened and postconditions are strengthened.

**Theorem 5.1 (Refinement Law)** *A qualitative action A is refined by a continuous action C, written $A \sqsubseteq_r C$, iff*

$$
\begin{aligned}
& [\ \neg\infty_q(Q, e_q, d_q) \wedge r \implies \neg\infty_c(X, e_c, d_c))\ ] \quad \textit{and} \\
& [\ (\neg\infty_q(Q, e_q, d_q) \wedge r \wedge (\exists \phi : C^1 \bullet SF_c(\phi, e_c, d_c) \wedge \Delta_c(\phi, e_c) > 0 \\
& \qquad \wedge X' = \phi.\Delta_c(\phi, e_c) \wedge \dot{X}' = \dot{\phi}.\Delta_c(\phi, e_c))) \\
& \qquad \implies \\
& \exists Q', \psi : QTS, g \in traces.\psi \bullet (SF_q(g, e_q, d_q) \wedge Q' = g.\Delta_q(g, e_q) \wedge r)\ ]
\end{aligned}
$$

*with $[\,]$ denoting universal quantification over the observations before $(X, \dot{X}, Q)$ and after execution $(X', \dot{X}', Q')$. The termination predicates are defined as:*

$$
\begin{aligned}
\neg\infty_q(Q, e_q, d_q) &=_{df} \forall \psi : QTS, g \in traces.\psi \bullet SF_q(g, e_q, d_q) \wedge \\
& \qquad \Delta_q(g, e_q) > 0 \implies \Delta_q(g, e_q) < \infty \\
\neg\infty_c(X, e_c, d_c) &=_{df} \forall \phi : C^1 \bullet SF_c(\phi, e_c, d_c) \wedge \Delta_c(\phi, e_c) > 0 \implies \Delta_c(\phi, e_c) < \infty
\end{aligned}
$$

**Proof 5.1.**

$$
\begin{aligned}
& A \sqsubseteq_r C \\
& \quad \equiv \quad \{L\ simulation, Lemma\ 5.1\} \\
& \{p_A\};[R_A];[r] \supseteq [r];\{p_C\};[R_C] \\
& \quad \equiv \quad \{wp\ of\ sequential\ composition\} \\
& wp(\{p_A\}, wp([R_A], wp([r], post))) \implies wp([r], wp(\{p_C\}, wp([R_C], post))) \\
& \quad \equiv \quad \{wp\ definitions\} \\
& \forall \sigma, \gamma \bullet wp(\{p_A\}, \forall \sigma' \bullet R_A.\sigma.\sigma' \wedge r.\sigma' \subseteq \gamma) \implies wp([r], \forall \sigma' \bullet p_C.\sigma' \wedge R_C.\sigma' \subseteq \gamma) \\
& \quad \equiv \quad \{wp\ definitions\} \\
& \forall \sigma, \gamma \bullet p_A.\sigma \wedge \forall \sigma' \bullet R_A.\sigma.\sigma' \wedge r.\sigma' \subseteq \gamma \implies (\forall \sigma' \bullet r.\sigma.\sigma' \implies p_C.\sigma' \wedge R_C.\sigma' \subseteq \gamma) \\
& \quad \equiv \quad \{\Rightarrow by\ specialization\ (\gamma := R_A;r), \Leftarrow by\ transitivity\ of\ \subseteq\} \\
& \forall \sigma, \sigma' \bullet p_A.\sigma \wedge r.\sigma.\sigma' \implies p_C.\sigma' \wedge R_C.\sigma' \subseteq R_A;r \\
& \quad \equiv \quad \{definition\ of\ \subseteq\} \\
& (\forall \sigma, \sigma' \bullet p_A.\sigma \wedge r.\sigma.\sigma' \implies p_C.\sigma') \wedge \\
& (\forall \sigma, \sigma', \gamma \bullet p_A.\sigma \wedge r.\sigma.\sigma' \wedge R_C.\sigma'.\gamma \implies \exists \sigma'' \bullet R_A.\sigma.\sigma'' \wedge r.\sigma''.\gamma) \\
& \quad \equiv \quad \{definitions\ Lemma\ 5.1\} \\
& [\ \neg\infty_q(Q, e_q, d_q) \wedge r \implies \neg\infty_c(X, e_c, d_c))\ ] \quad and \\
& [\ (\neg\infty_q(Q, e_q, d_q) \wedge r \wedge (\exists \phi : C^1 \bullet SF_c(\phi, e_c, d_c) \wedge \Delta_c(\phi, e_c) > 0 \wedge \\
& \qquad X' = \phi.\Delta_c(\phi, e_c) \wedge \dot{X}' = \dot{\phi}.\Delta_c(\phi, e_c))) \\
& \quad \implies \\
& \exists Q', \psi : QTS, g \in traces.\psi \bullet (SF_q(g, e_q, d_q) \wedge Q' = g.\Delta_q(g, e_q) \wedge r)\ ]
\end{aligned}
$$

□

This formal proof follows the refinement calculus style of Back and von Wright [18].

### 5.1.3. Testing

In contrast to work presented in the previous chapter, where whole trajectories are tested against specified traces, this approach builds on the weakest precondition semantics of differential/qualitative actions. This has as consequence, that only the pre/post states of actions are observable but not the trajectories/traces in between. Figure 5.2 shows a concrete trajectory and the according abstract trace through the example system consisting of four evolutions. For visualization we also show the behavior in between the pre/post states of evolutions.

The system may be initialized with both tanks being empty and both pumps turned off. (1) In the first evolution both pumps are turned off and the inflow *in* fills tank $T1$ up to the $Full$ level. (2) When $T1$ is full pump $P1$ is activated and delivers water to $T2$, hence the water level $x_2$ is increasing. (3) When the water level $x_1$ drops to *Empty* the pump $P1$ is turned off. Since $P2$ still is turned off, $x_2$ remains constant. Due to the inflow, the water level in tank $T1$ increases. (4) In the last evolution pump $P2$ is started which empties tank $T2$ below the *Reserve* water level.

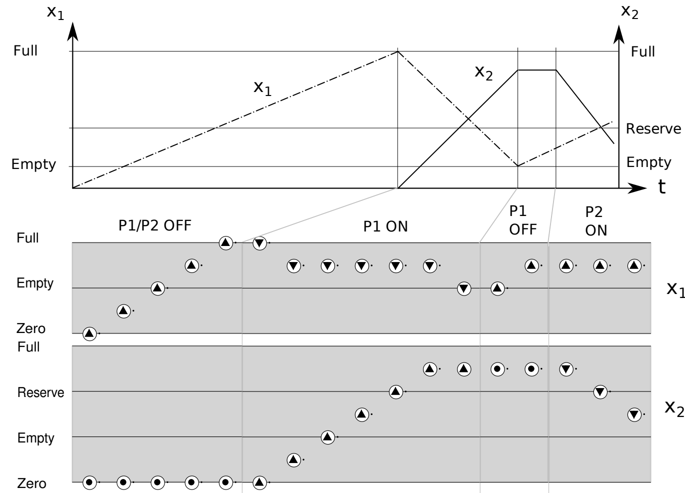



Figure 5.2.: Example Test Case consisting of four Evolutions.

The visible behaviors of the example system are the traces consisting of the valuation of model variables before and after action executions. In the next section we introduce a further level of abstraction by labeling all actions with names that may have parameters. The execution of a named action corresponds to an event and the set of all possible execution sequences, starting from the initial state, gives a QAS an LTS semantics. Thus, the valuation of the state variables becomes internal.

## 5.2. Automated Conformance Verification of Hybrid Systems

In order to apply model-based testing using labeled transition systems, actions are augmented with names (labels) which may have parameters. Unnamed actions are internal and, when executed, cause $\tau$ events. The set of labels $L$ is partitioned into inputs $L_I$ and outputs $L_U$. Then state space exploration of the action system yields an LTS which can be used for test case generation. In particular we derive mutation-based test cases by verifying the conformance between an original and a mutated specification QAS. The approach of fault or mutation based testing was presented in Section 2.1.5. The generation of mutation-based test cases is covered in Section 5.3.

Figure 5.3 shows the labeled QAS of our two-tank example. Discrete actions have the form $label : guard \rightarrow body$. The actions with guards $g_1$ to $g_4$ are controlled by the system and hence are observable by the tester. In order to control the system we have added a further discrete action $WATER_REQ(X)$ which is an input action to the system. The Boolean parameter $X$ determines if water is required or not, and the guard $g_5$ describes a simple user scenario: A user turns on the water when the tank $T2$ is full and turns it off again when the water drops below the reserve level.

$$g_5 =_{df} (\neg wr \wedge x_2 = Full \wedge X = true) \vee (wr \wedge x_2 < Reserve \wedge X = false)$$

The imported quantity $in$ represents the inflow into tank $T1$. As we do not define an external system which controls the inflow we set it to some constant rate, $in =_{df} (0..Max, 0)$. Hence, the system is closed and the inflow is actually part of the state variables.

$$
\begin{array}{llll}
QSystem =_{df} |[ & \textbf{var} & & x_1 : T1, x_2 : T2, out, inout : FR \\
 & & & diff_1, diff_2 :: NZP \\
 & & & P1_running, P2_running : Bool \\
 & & \bullet & x_1 := (0,0); x_2 := (0,0); out := (0,0); inout := (0,0); \\
 & & & P1_running := false; P2_running := false; wr := false \\
 & \textbf{alt} & & PUMP1_ON : g_1 \rightarrow p1_running := true; inout := (0..Max, 0) \\
 & & \Box & PUMP1_OFF : g_2 \rightarrow p1_running := false; inout := (0,0) \\
 & & \Box & PUMP2_ON : g_3 \rightarrow p2_running := true; out := (0..Max, 0) \\
 & & \Box & PUMP2_OFF : g_4 \rightarrow p2_running := false; out := (0,0) \\
 & & \Box & WATER_REQ(X) : g_5 \rightarrow wr := X \\
 & \textbf{with} & & \neg(g_1 \vee g_2 \vee g_3 \vee g_4 \vee g_5) :\rightharpoondown \\
 & & & add(diff_2, out, inout) \wedge add(diff_1, inout, in) \wedge \\
 & & & d/dt(x_1, diff_1) \wedge d/dt(x_2, diff_2) \\
]| & : in & &
\end{array}
$$

Figure 5.3.: Labeled QAS of the Two-Tank System.

Notice, that qualitative actions have no name as they are treated in a different manner than input and output actions. Since the QDEs of qualitative actions are an abstract representation of ODEs, inputs and outputs evolve in parallel over time. The continuous behavior is a vector of two functions $[in(t), out(t)]^T$ where *in* and *out* evolve simultaneously. Hence, for qualitative actions we introduce a special label *qual* which may have parameters. A *qual* event denotes the end of an evolution and assigns to possible parameters the according values of qualitative state variables. For a tester *qual* events are observable, or internal if they are not visible at the testing interface.

### 5.2.1. On-the-fly Conformance Checking

A conformance relation is defined between two formal models. Usually one of the models is the implementation model, however we determine the conformance between two specifications. These specifications represent the behaviors of hybrid systems which are derived on-the-fly during the exploration of our specification models, i.e. labeled QAS.

The labeled transition system of a QAS $M$ is given by considering action names as labels in the LTS. Then the LTS semantics of a QAS $M$ is given by the set of traces $M$ can produce after a finite number of computation steps. More formally: Given a QAS $M$, and an initial state $s_0 \in S$, the set of traces is defined as:

$$
\begin{aligned}
traces(M) =_{df} \{\sigma \downarrow L \mid (\sigma \in (L \cup I)^* \bullet \\
\sigma = \langle a_0, \ldots, a_n \rangle \wedge s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_n \xrightarrow{a_n} \wedge \forall 0 \leq i \leq n \bullet g(a_i)(s_i))\}
\end{aligned}
\tag{5.13}
$$

Here, $g(a_i)$ is the enabledness guard of action $a_i$ and $I$ denotes the set of internal actions. If the current state $s_i$ satisfies the enabledness guard the according action is executed and the event

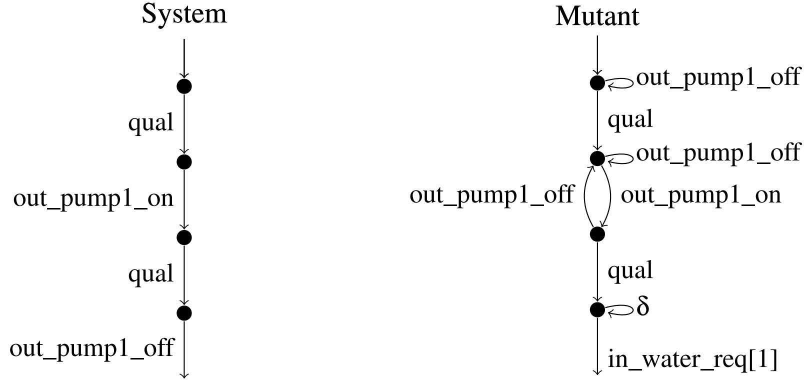



Figure 5.4.: Two suspension automata showing the behavior of the water tank example and of a mutated version.

name $a_i$ is appended to the trace which led to $s_i$. Furthermore, the action execution updates state $s_i$ to $s_{i+1}$. Note, that internal actions, when enabled, cause $\tau$ transitions: $\forall i \in I, s \in S \bullet g(i)(s) \implies s \xrightarrow{\tau}$. Since we are only interested in traces over the alphabet $L$, internal actions are removed by projecting $\sigma$ onto $L$, see (5.13).

We explore the LTS of a QAS by forward execution starting from a given initial state. In order to execute a guarded command $p \rightarrow A$ its enabledness $g$ guard must hold:

$$g(p \rightarrow A) \equiv \neg wp(p \rightarrow A, false) \equiv \neg(p \implies wp(A, false)) \equiv p \wedge g(A).$$

In terms of enabledness, a nondeterministic demonic choice between two action becomes an angelic choice for forward execution:

$$\neg wp(A \Box B, false) \equiv \neg(wp(A, false) \wedge wp(B, false)) \equiv g(A) \vee g(B).$$

A demonic choice between two actions means that it must be ensured that both actions establish the post condition, no matter which one is chosen. This corresponds to the conjunction of the weakest preconditions. For an angelic choice at least one action has to satisfy the post condition which corresponds to the disjunction of weakest preconditions.

**Example 5.1.** Figure 5.4 shows two suspension automata obtained from the system specification, see Figure 5.3, and from a mutant. A mutant is a faulty version of the original. The "in_" and "out_" prefixes in the LTSs denote input and output events respectively, *qual* is per definition an output event. One can observe that the mutant does not conform with respect to ioco because $out(Mutant\ \textbf{after}\ \langle\rangle) = \{qual, out_pump1_off\} \not\subseteq \{qual\} = out(System\ \textbf{after}\ \langle\rangle)$.

Let us consider the generation of the LTS "System" from the specification QAS. Starting from the initial state only the qualitative action is enabled. This is because all guards ($g_1, \ldots, g_5$) evaluate to false in the initial state. The execution of the action leads to the *qual* event after the initial state. The qualitative action updates $x_1$ from $(0,0)$ to $(full, 0)$. In this state the guard $g_1$ evaluates to true. Hence, the action *PUMP1_ON* can be executed. This results into the *out_pump1_on*-labeled transition in the LTS. The other parts of the system are derived in a similar manner. For

the sake of brevity, we do not show the complete LTS of the system. Note, that the Mutant LTS shows parts of an LTS obtained by negating the guard $g_2$ in the original specification. Also here only parts of the whole LTS are shown. □

The conformance verification between two LTSs, i.e $LTS_1$ *ioco* $LTS_2$, is achieved by computing the synchronous product modulo the conformance relation [163], i.e. $LTS_1 \times_{ioco} LTS_2$. Here, $LTS_1$ is the implementation. In the case of non-conformance the resulting product LTS will contain special *fail* states. The conformance verification is then achieved by a reachability analysis of *fail* states. Any trace leading to a fail state is a counter-example showing non-conformance. The conformance is verified up to a defined search depth. A mutant is called equivalent regarding the search depth if it shows no deviating behavior.

Informally, the synchronous product $LTS_1 \times_{ioco} LTS_2$ is calculated by applying one of the following rules:

- Transitions that are possible in both LTSs are transitions of the synchronous product.
- Inputs that are allowed in $LTS_1$ but are not allowed in $LTS_2$ lead to a sink state labeled with *pass*. This rule reflects the fact that implementations may behave arbitrarily after unspecified inputs.
- For any output that is allowed in $LTS_2$ but not in $LTS_1$ a transition to a pass state is added.
- For inputs that are enabled in $LTS_2$ but not in $LTS_1$ the labeled transition system $LTS_1$ is made input-enabled, i.e. such transitions are part of the synchronous product.
- If an output of the left hand LTS $LTS_1$ is not an output of the right hand LTS $LTS_2$ a transition leading to a *fail* state is added to the synchronous product.

We have developed the prototype tool *Ulysses* written in SICStus Prolog[2] which explores two given QAS while checking conformance on-the-fly. The name for the tool originates from the similarity between the search in the state space for *fail* states and an odyssey. For solving the qualitative differential equations we employ the tool *ASIM* [22]. Remember, *ASIM* is a Prolog implementation of the QSIM algorithm. We have ported *ASIM*, which is originally implemented in GNU Prolog[3], to SICStus Prolog.

**Example 5.2.** The calculation of the synchronous product between the two labeled transition systems of Figure 5.4 leads to the LTS shown in Figure 5.5 (generated with *Ulysses*). For example, the *qual*-action is enabled in both LTS, thus it is part of the synchronous product. The output action *out_pump1_off* is not allowed by the specification. Consequently, this action leads to a fail state in the synchronous product. After the *qual*-action the *out_pump1_off* action is still not allowed by the specification. Thus, again this action leads to a fail state. The 'delta' label denotes quiescence, i.e. the absence of any observations, see Section 2.2.2. Note, that Figure 5.5 shows the synchronous product for the larger LTSs than those depicted in Figure 5.4. □

The product calculation is performed on-the-fly. Ulysses explores two given Qualitative Action Systems state by state and computes the synchronous product according to the ioco product

[2] www.sics.se/sicstus
[3] www.gprolog.org

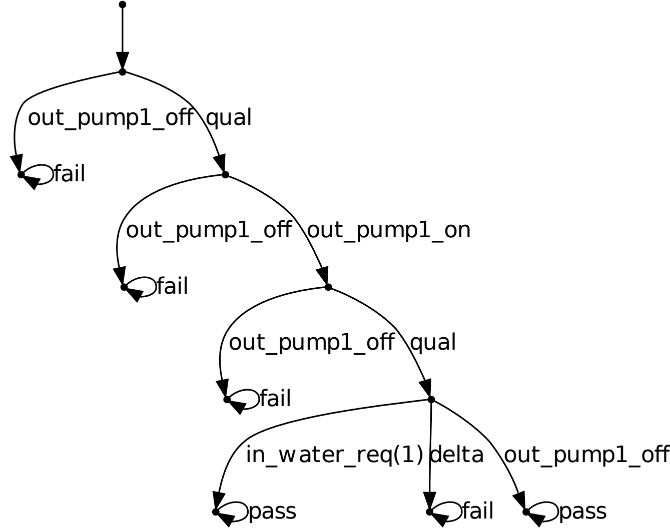



Figure 5.5.: Conformance verification result with a mutated action 'out_pump1_off'.

rules. On-the-fly computation can be quite efficient when traces which reveal the fault are rather short. However, in the worst case the complete specification has to be unfolded during product calculation.

**Algorithm 5.1** $getSuccessors(qas, s_1) : L \cup \{\delta\} \mapsto \mathcal{P}(S)$

1: $s_2 := tauClosure(R, s_1, \tau);$
2: $qstates := s_2 \setminus \{s \in s_2 \mid \exists ev \in L_U \cup \tau, s' \in \mathcal{P}(S) \bullet s' = R(s, ev)\};$
3: $succ := [ev \mapsto \{s \mid \exists s' \in s_2, s'' \in \mathcal{P}(S) \bullet s'' = R(s', ev) \wedge s \in s''\} \mid ev \neq \tau];$
4: **if** $qstates \neq \emptyset$ **then**
5: $\quad succ := succ \cup [\delta \mapsto qstates];$
6: **end ifreturn** $succ;$

Algorithm 5.1 computes the successor events and successor states in the suspension automaton $\Gamma$ of a given QAS. Because of determinization via subset construction the states in $\Gamma$ are sets of states. Given a specification *qas* and a state $s_1$ *getSuccessors* returns a map from events to successor states. For the successor computation we apply a relation $R(s, ev)$ which takes a state $s$ and an event $ev$ and calculates the successor states of $s$ that are reachable by event $ev$. More formally,

$$R(s, ev) = \{s_1 \mid \exists s_2 \bullet int(qas, s, s_2, ev) \wedge s_1 \in s_2\}$$

Here, the predicate $int : QAS \times S \times \mathcal{P}(S) \times (L \cup \{\tau\}) \mapsto bool$ interprets the given QAS specification by nondeterministically interpreting all actions.

First, in Line 1 all states reachable via internal actions are computed. This process is called

$\tau$-closure computation. We assume that specifications are strongly converging, i.e. there are no infinite sequences of internal computations. Because of its pattern matching and backtracking capability Prolog is well suited to describe search and exploration problems. Listing 5.1 shows that *tauClosure* can be written quite compact with only three Prolog clauses.

Listing 5.1: tauClosure

```
1 tauClosure(QAS, S1, S2)     :- tauClosure(QAS, S1, S1, S2).
  tauClosure(_, [], Cl, Cl)   :- !.
3 tauClosure(QAS, S1, A, Cl)  :-
      findall(S2, (member(S, S1), int(QAS, S, S2, tau)), S3),
5     flatten(S3, States),
      difference(States, A, Frontier),
7     union(Frontier, A, A1),
      tauClosure(QAS, Frontier, A1, Cl).
```

Here, the predicates *difference(A, B, C)* and *union(A, B, C)* are the conventional set operations $C = A \backslash B$ and $C = A \cup B$, respectively. When an action is enabled the interpreter executes it and returns its label plus the set of successor states. In the current version of our tool actions are interpreted with concrete values, thus nondeterministic updates of state variables yield a set of successor states.

For the purpose of $\tau$-closure computation the event variable of the *int* predicate is instantiated with `tau`. This ensures that only internal actions are interpreted. The built-in predicate *findall* collects all successor states $S2$ of executed internal actions, that are enabled in state $S$, in a list stored in variable $S3$. The *member* predicate enumerates all states $S \in S1$. Since $S2$ is a list of states, $S3$ is a list of lists. In Line 7 the built-in predicate *flatten* converts $S$ into a single list of states $States$. The set difference in *Frontier* contains new states to be explored and the set union in $A1$ are the states which have already been visited. When the base case in Line 3 is reached the exploration agenda in *Frontier* is empty. Then the variable $S2$ in Line 1 is unified with the list of all states reachable by the execution of internal actions.

Coming back to the description of Algorithm 5.1, the set of states after $\tau$-closure computation, from which no internal or output action is enabled, are *quiescent* states and stored in *qstates* (Line 2). Next, the specification is interpreted for all states $s' \in s2$. All successor events $ev \neq \tau$ and successor states $s$ are entered into the successor map $succ : L \cup \{\delta\} \mapsto \mathcal{P}(S)$ via the map comprehension in Line 3. If there are some quiescent states then the successor map is extended with a $\delta$ event associated with the set of quiescent states (Line 5). Finally, the successor map is returned by *GetSuccessors*.

During exploration the interpretation of qualitative actions requires to solve QDEs. When the evolution guard of a qualitative action holds in a certain state then the QR tool *ASIM* [22] is called to compute the qualitative transition system. Then, according to Definition 5.5, a breadth-first search determines the set of states where the evolution terminates. From these states the exploration of further actions proceeds.

Starting from an initial state and by recursively applying the *getSuccessor* algorithm to two given specifications we compute the synchronous product by following the rules described above.

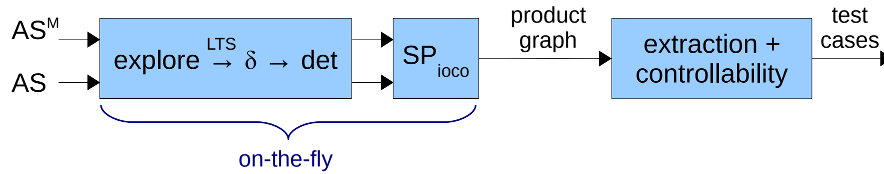



Figure 5.6.: Computation steps of Ulysses.

## 5.3. Mutation-based Test Case Generation

For test case generation we follow a mutation-based approach: test cases are derived from the discriminating behavior regarding a conformance relation between an original and a mutated specification. Each mutant contains exactly one syntactic manipulation of the original specification. Equivalent mutants are recognized and ignored for test case generation.

We prefer the term conformance checking to equivalence checking because of the fact that our models are non-deterministic. Therefore, conformance is formally a pre-order relation, but not an equivalence relation. For testing we are interested in partial system models. This restricts the choice of conformance relations to those that only consider the traces of the specification rather than the complement of traces of the implementation. Furthermore, we want to apply active testing by sending inputs to and observing outputs from the implementation, and we want to check if quiescence is allowed in certain states. Considering these requirements the *ioco* relation is the right choice.

Figure 5.6 depicts the computation steps of Ulysses. The tool expects two labeled action systems as input: (1) a system specification *QAS* and (2) a mutated version of the same specification $QAS^M$. The explored LTSs are augmented with quiescence transitions and subsequently converted into a deterministic automaton. By executing these steps, which are depicted in the first box of Figure 5.6, we obtain a so-called *suspension automaton*.

Ulysses generates the suspension automata for both input models *QAS* and $QAS^M$. Afterwards, the *ioco* check for these two models is performed (see the central box in Figure 5.6), which generates a product graph. From this graph we extract controllable test cases (last box in the *Ulysses* process). Note that the calculation of the suspension automata and the synchronous product calculation modulo *ioco* ($SP_{ioco}$) are performed on-the-fly, which means that the automata are only unfolded as required by the conformance check.

### 5.3.1. Ensuring Controllability in Presence of Non-determinism

The *ioco* relation is a global property referring to traces of the specification rather than to states like simulation relations. This implies that the local information in a certain state, in general, is not enough to decide conformance. That is the case for non-deterministic specifications where the same trace leads to different states. Here the outputs of both states have to be considered by *ioco* which requires preceded determinization. Ulysses applies the computation of the suspension automaton on-the-fly while determining the product LTS of two given QAS. However, care has

to be taken during determinization regarding the enabledness of events. In action systems, events can only occur if the action's enabledness guard is satisfied in the current state. This guard ensures that the action will not terminate in an undefined post-state.

In the case of black-box testing, the state of a system is internal and cannot be observed from outside. If an action system model contains non-determinism, i.e., internal actions and nondeterministic updates, determinization of its labeled transition system via subset construction leads to sets of states. In the action system model, however, for each of these states different subsequent events (actions) could be enabled. Due to the abstraction to an LTS we lose this state information and in the LTS the union of all enabled events would be indicated as valid subsequent actions. This information loss leads to a problem during test case execution, as the implementation might make another internal decision than the test driver. In the end, the tester might not know in which state the SUT is in and worse, which events are allowed. Because we require the SUT to be input-enabled, the tester might not even notice the loss of synchronization immediately: the SUT has to ignore all inputs that are not allowed. Of course, if the tester subsequently encounters an unexpected observation, it would issue a fail verdict - which would be wrong in this case, as the SUT made a valid internal choice.

To overcome this problem, we need to synchronize the test case execution with the internal state decisions of the implementation. Since this is not possible in blackbox testing, the alternative is to disallow input events with guards that do not hold for all internal states. We denote the occurrence of an event $a$ in a state $s$ as $s \xrightarrow{a}$. The following definition states that an action from the set of input actions $L_I$ is enabled in a state $s \in S$ of the suspension automaton iff the action is enabled in all sub-states of $s$:

$$\forall a \in L_I, s \in S \bullet (\forall s_i \in s \bullet g(a)(s_i)) \iff s \xrightarrow{a} \tag{5.14}$$

According to *ioco* this is a valid abstraction in the sense of less choices in the environment, i.e. inputs can be removed. This leads to a reduction in the number of testing scenarios. If this causes the loss of many input events then this is an indication that the tester needs more observations to control the implementation. The work in [164] exploits the observation of quiescence to resolve the internal decision of the implementation to either accept inputs or to produce outputs. The enabledness of (additional) output events causes no problem since the *ioco* relation allows weaker output behavior in the specification. For events in the output alphabet $L_U$ we define:

$$\forall a \in L_U, s \in S \bullet (\exists s_i \in s \bullet g(a)(s_i)) \iff s \xrightarrow{a} \tag{5.15}$$

In conclusion, the suspension automaton is a valid abstraction of the event traces generated during the exploration of an action system regarding *ioco*.

**Example 5.3.** Consider an internal choice between two events, i.e., $(\tau; ?a) \sqcap (\tau; !b)$, where $\tau$ denotes an internal event, $?a$ is an input event, $!b$ is an output event, the semicolon denotes sequential composition, and $\sqcap$ is the nondeterministic choice. The LTS 1 in Figure 5.7 shows the LTS of the internal-choice example. Notice, that states with output quiescence are augmented with $\delta$ self loops. The suspension automaton in LTS 2 is obtained after determinization of LTS 1 by computing the $\tau$ closure. A problem arises here since both events $?a$ and $!b$ can occur at the

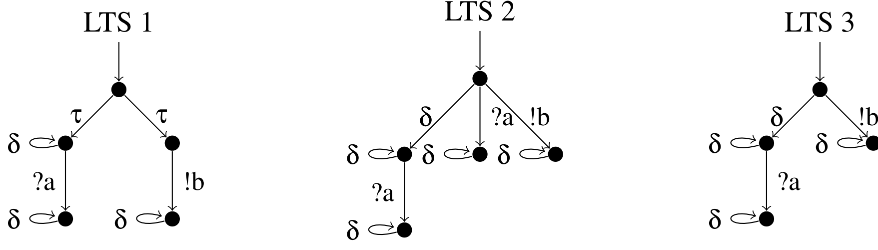



Figure 5.7.: Internal choice between an input and an output action.

initial state which is not the behavior specified by the internal choice. We can only apply input $?a$ if the implementation is in the according internal state. Hence, we forbid event $?a$ in the initial state, see LTS 3. Then two things can happen: either the output event $!b$ or no output is observed. In the case of quiescence, the tester changes to a state where the input event is enabled. □

Due to the inherent non-determinism of qualitative actions, controllability is a major issue. It may occur that a tester loses controllability over the IUT if there are not enough observations for the tester to synchronize. In such a case the introduction of additional observations is required to resolve the problem.

### 5.3.2. Test Case Selection

In the following we first show the required properties of our selected test cases and then present a selection algorithm for adaptive test cases.

Given a product LTS $(S^P, L^P, T^P, s_0^P)$ being the result of an *ioco* check and a set $Fail \subset S^P$ denoting the set of fail states. Note, that fail states in the product LTS denote special fail states that reflect the fault in the model. They are used as test goal however, since fail states are implicit they are not part of a test case. We define an unsafe state as a state after which we may fail.

$$Unsafe =_{df} \{s \in S^P \mid \exists a \in L_U \bullet s \xrightarrow{a} f \wedge f \in Fail\} \qquad \text{(unsafe states)}$$

Unsafe states play the central role in our test case generation strategy, since they are the test goal a test case should cover. Below, we present the general properties of a generated test case LTS $TC = (S^{TC}, L^{TC}, T^{TC}, s_0^{TC})$ that is selected from a product.

$$S^{TC} \setminus \{pass, inconc\} \subseteq S^P \wedge L^{TC} \subseteq L^P \wedge s_0^{TC} = s_0^P \qquad \text{(inclusion)}$$

$$s \not\xrightarrow{a} \iff s \in \{pass, inconc\} \qquad \text{(sink states)}$$

$$s = pass \iff \exists u \in Unsafe, a \in L_U \bullet u \xrightarrow{a} s \wedge s \notin Fail \qquad \text{(passing)}$$

$$(s \in S^{TC} \wedge s \xRightarrow{a} \wedge s \xRightarrow{b}) \implies (a, b \in L_I \wedge a = b) \vee a, b \in L_U \qquad \text{(controllability)}$$

$$\exists!\, u \in S^{TC} \bullet (u \in Unsafe) \wedge \exists a \in L_U \bullet u \xrightarrow{a} pass \qquad \text{(test goal unique)}$$

$$\exists\, \sigma \bullet s_0^{TC} \xRightarrow{\sigma} u \wedge u \in Unsafe \qquad \text{(test goal reachable)}$$

A test case is a sub-transition system of the calculated product extended with two additional verdict states *pass* and *inconc* (controllability). In the test case, a sink state is a verdict state *pass* or *inconc*. Note, fail verdicts are implicit and not included (sink states). The pass verdict is characterized by successfully passing an unsafe state (passing). A test case does not contain choices over controllables, see also [156] (controllability). According to the work in [156] controllability also prohibits the occurrence of inputs and outputs in a state.

Reaching an unsafe state is the test goal of our mutation testing strategy. Hence, per test case exactly one unsafe state precedes a pass verdict state (test goal unique). In highly nondeterministic specifications it may make sense to generate test cases due to several or all unsafe states in a product graph. This increases the change of reaching the test goal and hence reduces the number of inconclusive verdicts. Finally and most importantly, the test case must be able to reach its test goal, i.e., the unsafe state (test goal reachable).

There are two kinds of test cases that may reach our test goal, i.e., a given unsafe state: First, a linear test case that includes one path to the unsafe state. Second, a branching adaptive test case that may include several paths to an unsafe state. In the following, we discuss the properties of the two kinds.

**Linear Test Cases** are necessary if the target test harness does not support branching behavior. The following two additional properties characterize linear test cases:

$$\exists!\, \sigma \bullet s_0^{TC} \stackrel{\sigma}{\Rightarrow} u \wedge u \in \mathit{Unsafe} \qquad \text{(linear test case)}$$

$$s^{TC} \stackrel{a}{\rightarrow} \mathit{inconc} \iff s^{TC} \stackrel{\sigma}{\Rightarrow} u \wedge u \in \mathit{Unsafe} \wedge \exists s' \bullet ((s^{TC}, a, s') \in T^P \wedge a \in L_u \wedge a \notin hd(\sigma)) \qquad \text{(inconclusive linear)}$$

Such a test case contains exactly one path to the unsafe state (linear test case). Since a model's behavior may branch, an observation may lead away from the linear path. In this case, the test has to be stopped with an inconclusive verdict (inconclusive linear). Here, the head operator *hd* returns the first element in the trace $\sigma$.

**Adaptive Test Cases** integrate several paths to the unsafe state into one test case. They only give an inconclusive verdict if it is impossible to reach the unsafe state:

$$s^{TC} \stackrel{a}{\rightarrow} \mathit{inconc} \iff s^{TC} \stackrel{\sigma}{\Rightarrow} u \wedge u \in \mathit{Unsafe} \wedge \exists s' \bullet ((s^{TC}, a, s') \in T^P \wedge a \in L_u \wedge s' \stackrel{\sigma'}{\not\Rightarrow} u) \qquad \text{(inconclusive adaptive)}$$

Note the difference to the linear test case in the last conjunct. A linear test case reports inconclusive if an observation leads away from the single path to the unsafe state. In contrast, in the adaptive case, inconclusive is only reported, if after an observation we are unable to reach the unsafe state.

For the generation of adaptive test cases we apply a shortest-trace based search strategy. We define a trace as a sequence of labels which can be traversed along transitions between two nodes in the product graph. Definition 5.16 states the set of traces between a start node and a set of destination nodes in a graph, in our case the product LTS. After having the notion of *traces*

Definition 5.17 denotes a shortest trace between a start and a set of destination nodes. Note, that there can exist several shortest traces having the same length. Since we are only interested in an arbitrary shortest trace we choose one nondeterministically.

$$traces(s_a, S_b) =_{df} \{\sigma \in L^* \mid \exists s_b \in S_b \bullet s_a \stackrel{\sigma}{\Rightarrow} s_b\} \tag{5.16}$$

$$shortestTrace(s_a, S_b) =_{df} \sigma \in traces(s_a, S_b) \mid \forall \sigma' \in traces(s_a, S_b) \bullet |\sigma| \leq |\sigma'| \tag{5.17}$$

**Algorithm 5.2** $getTC(s, Goal) : \mathcal{P}(S \times L \cup \delta \times (S \cup inconc))$

```
1: if s ∈ Goal then
2:     return ∅
3: else
4:     σ := shortestTrace(s, Goal)
5:     if σ = ⟨⟩ then
6:         return {(s, a, inconc) | ∃s' • s -a-> s' ∧ a ∈ L_U ∧ s' ∉ Fail}
7:     else
8:         E := edges(σ)
9:         Goal := Goal ∪ states(E)
10:        return E ∪ ⋃{{(s1, a, s2) ∈ T^P} ∪ getTC(s2, Goal) |
                  ∃a', s2' • (s1, a', s2') ∈ E ∧ a, a' ∈ L_U ∧ a ≠ a' ∧ s2 ∉ Fail}
11:    end if
12: end if
```

Algorithm 5.2 describes the selection of adaptive test cases. In the following we denote set variables with capital letters. Given a product LTS and an unsafe state $u$, we obtain an adaptive test case $TC_u$ by calling $getTC(s_0, \{u\})$. A test case is a selection of transitions in the product LTS plus transitions to the test verdict states *pass* and *inconc*. Hence, a test case has the type $\mathcal{P}(S \times (L \cup \delta) \times (S \cup pass \cup inconc))$. By searching for the shortest trace from the initial state to an *unsafe* state and recursively extending this trace for all branching output transitions, the obtained test case complies with the controllability requirement (controllability).

In Line 1 it is checked if the current state is already a goal state. If it is a goal state the empty set is returned. In Line 4 the shortest trace between a state $s$ and one state in the set of goal states is determined. Note that there can exist several shortest traces of same length. In this case, we non-deterministically choose one. If there exists no such trace then all observations in state $s$ terminate in an inconclusive state. Otherwise the transitions $E$ along trace $\sigma$ (Line 8) are part of the test case. The transitions of a trace are determined by the function: $edges : (L \cup \delta)^* \mapsto \mathcal{P}(S \times (L \cup \delta) \times S)$.

In Line 9, the set of states in $E$ obtained by the function *states* are added to the set of goal states. Hence, the goal becomes to reach one state of the test case which in turn leads to an unsafe state. Next, the test case is recursively extended by all branching observable events along the trace $\sigma$ (Line 10). Here, the set comprehension creates a set of sets of transitions. Each set corresponds to a subgraph in the test case after a branching observation. Here, $a'$ are the labels on the shortest path $\sigma$. The recursive algorithm terminates when all branching outputs have been processed. Finally, the graph obtained from *getTC* is extended with the transitions to the pass state, see Definition (adaptive test case).

$$TC_u =_{df} getTC(s_0, \{u\}) \cup \{(u, a, pass) \mid \exists s \bullet u \xrightarrow{a} s \ \wedge a \in L_U \ \wedge s \notin Fail\}$$

(adaptive test case)

**Example 5.4.** Figure 5.8 shows the conformance verification result between the system specification and a mutant. The shaded states in the figure depict a test case which was created with our adaptive test case algorithm. For presentation purposes we have applied *strong bisimulation* minimization, using the *CADP* toolbox[4], to the product LTS obtained from Ulysses. In the mutant, the guard of the action $WATER_REQ(X)$ has been set to *false*. As can be seen, the mutation is revealed when the tester tries to turn on pump $P2$ via a water request. According to Definition (unsafe states) State 6 is an unsafe state. By coincidence state Number 2 is a pass state in the product as well as in the test case. However, these two different types of sink states should not be confused. A pass state in the product denotes the end of the specified behavior. This occurs when the implementation (in our case the mutant) applies an unspecified input or produces fewer outputs than specified. A pass state in the test case denotes the achievement of the test goal, i.e. to pass an unsafe state. Notice, that for each label on the edges between State 15 to State 16 and State 9 to State 16 there exists a corresponding labeled transition.

By applying Algorithm 5.2 to the product LTS we obtain the test case which consists of the shaded states and according transitions between them. We start with the shortest trace $\sigma$ from the initial to the unsafe state, i.e. one trace over the sequence of states $\langle 0, 14, 9, 16, 8, 6 \rangle$. Next, we complete all states along trace $\sigma$ which have an outgoing observable event with all other observable events leaving these states. This extends the test case with transitions to the states $13, 5$, and 10. Then the algorithm is recursively applied to these new states. From state 10 we cannot reach the test goal, hence it is marked *inconclusive*. From state 13 there is exactly one trace back to the test case, thus this branch is closed. The recursion in state 5 adds transitions between states $5, 3, 7, 15$, and 16. Finally, in state 15 two transitions to states 5 and 10 are added which completes the test case.

The resulting test case is able to adapt to different parameterizations of the given system. For instance the volumes of both tanks or the flow rates of pumps may vary. Consider the example trace of the test case executed on a concrete implementation in Figure 5.9. Here, after turning on pump $P1$ twice the upper tank $T2$ has been filled which enables the activation of pump $P2$ satisfying the test goal. Observe, that the continuous evolution stops in state 16 since the execution of the last three discrete actions is considered to consume no time. During test case execution the state sequence $\langle 0, 14, 9, 5, 3, 7, 15, 16, 8, 6, 2 \rangle$ is visited until reaching the *pass* verdict.

There may also exist implementations where the upper tank is not completely filled after the last pumping phase of $P1$. In such a case the test case execution would end up in the inconclusive state. Furthermore, it may require several pumping phases of $P1$ such that the water level in tank $T2$ crosses a landmark value to the next interval. This is reflected by the two loops in states $\{14, 9, 13, 12\}$ and $\{5, 3, 7, 15\}$ respectively. □

Table 5.2.: Results when applying conformance verification to mutated specifications.

| Mut. Op. | No. Mutants | Avg.Time [s] | Average No. States | Average No. Trans. | $\neq$ | $=$ No. | $=$ Perc. |
|---|---|---|---|---|---|---|---|
| ASO | 10 | 13.9 | 64 | 117 | 7 | 3 | 30% |
| ENO | 6 | 7.6 | 68 | 120 | 5 | 1 | 17% |
| ERO | 20 | 12.9 | 62 | 110 | 20 | 0 | 0% |
| LRO | 13 | 12.8 | 93 | 168 | 9 | 4 | 31% |
| MCO | 16 | 12.8 | 70 | 126 | 10 | 6 | 38% |
| RRO | 12 | 12.0 | 40 | 73 | 10 | 2 | 17% |
| Total | 77 | 12.0 | 66 | 119 | 61 | 16 | 21% |

### 5.3.3. Experimental Results

Applying the conformance verification step to several different mutants yields the results shown in Table 5.2. We manually applied some well known mutation operators [28] to the QAS specification. In the first column *ASO* stands for the *Association Shift Operator* which changes the association between variables in Boolean expressions. *ENO* is the shorthand for the *Expression Negation Operator*, *ERO* means the *Event Replacement Operator*, *LRO* stands for *Logical Operator Replacement*, *MCO* denotes the *Missing Condition Operator*, and *RRO* is the abbreviation for the *Relational Replacement Operator*. The second column shows the number of generated mutants for each of the different operators. The average time needed for the conformance verification is given in the third column. The average number of states and transitions of the resulting product graphs are given in the fourth and fifth column, while the next to last column shows how many equivalent mutants were found: from a total of 77 mutants, 16 (about 21%) were found to be equivalent and cannot contribute any test cases. All conformance results were derived using unbounded search, i.e., the results are exact. The state space of the original specification comprises 59 states and 107 transitions.

[4] http://www.inrialpes.fr/vasy/cadp

fail 1 delta 6 out_pump2_on 2 pass
out_pump2_on
in_water_req(1)
out_pump1_off
15
out_pump1_on
7
8 delta
4
qual([t1:empty..full/std,t2:full/inc])
qual([t1:full/inc,t2:full/inc])
qual([t1:full/std,t2:full/inc])
qual([t1:empty/std,t2:full/inc])
qual([t1:empty..full/inc,t2:full/inc])
qual([t1:empty..full/dec,t2:full/inc])
qual([t1:empty/dec,t2:full/inc])
out_pump1_off
in_water_req(1)
11 delta
qual([t1:full/inc,t2:empty..reserve/std])
qual([t1:zero..empty/dec,t2:empty..reserve/inc])
16
qual([t1:zero..empty/dec,t2:reserve..full/inc])
out_pump1_off
3
10 inconc
qual([t1:empty..full/dec,t2:full/inc])
qual([t1:empty/dec,t2:full/inc])
qual([t1:full/inc,t2:full/inc])
qual([t1:empty/std,t2:full/inc])
qual([t1:empty..full/std,t2:full/inc])
qual([t1:full/std,t2:full/inc])
qual([t1:empty..full/inc,t2:full/inc])
out_pump1_off
5
qual([t1:zero..empty/dec,t2:reserve..full/inc])
qual([t1:zero..empty/dec,t2:empty..reserve/inc])
13
9
qual([t1:zero..empty/dec,t2:zero..empty/inc])
out_pump1_off
out_pump1_on
12
14
qual([t1:full/inc,t2:zero..empty/std])
qual([t1:full/inc,t2:zero/std])
0

Figure 5.8.: Product LTS and test case (transitions between shaded states depict the test case).

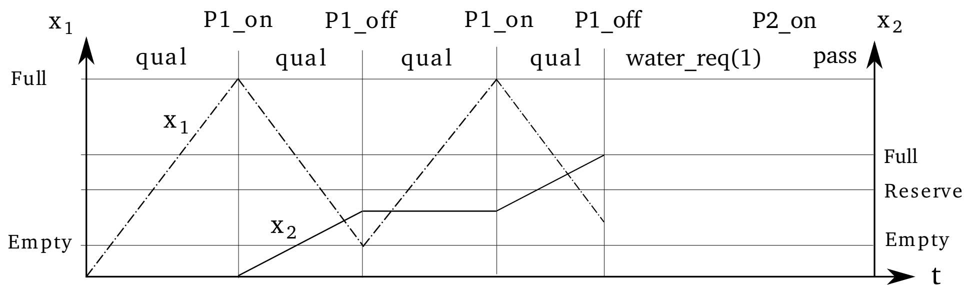



Figure 5.9.: Example execution of the test case.

Chapter 6

# Generation of Efficient Test Suites

*Parts of this chapter have been published in Aichernig et al. [9] and Aichernig et al. [10].*

A major part of this thesis was developed in the European FP7 project MOGENTES. The acronym stands for *Model-based Generation of Tests for Dependable Embedded Systems* and the aim of the project was to investigate methods for efficient test case generation as well as the applicability of such methods in industry.

Since the industry partners in the project employ the Unified Modeling Language (UML) in their work flows it was agreed to generate test cases from UML models. There is already existing work for test case generation from UML state charts, see [78] and [106]. However, these approaches do not fully support the required language features for our specification models. Additionally, we want to apply the well established input-output conformance relation for generated test cases. Therefore, our project partner Austrian Institute of Technology (AIT) developed with our collaboration an UML to action system translator. In particular UML state machines and class diagrams with OCL (Object Constraint Language) annotations are converted to so-called Object Oriented Action Systems (OOASs) [30]. These intermediate OOAS models are then flattened by the tool *Argos*, which was also developed within the MOGENTES project, into conventional action systems, see the work in [110]. Figure 6.1 depicts the complete tool chain for generating mutation-based test cases from UML models. Starting with an original and a mutated UML model the tool chain generates test cases due to certain test selection strategies.

The semantics of the UML language is not formally defined and leaves room for interpretations. The standard itself refers to such ambiguities as semantic variation points. Our translation scheme from UML to action systems gives UML a formal semantics suited for specifying our test models.

Since QAS models are an extension of conventional action systems (they contain an additional section for specifying qualitative actions) we can use Ulysses to generate test cases for purely discrete systems as well. Most of the project demonstrators are discrete systems or at least can be modeled as such without the need for stating qualitative actions. This chapter discusses the test case generation for object-oriented systems and approaches how to minimize test suites while

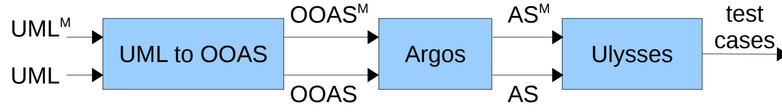



Figure 6.1.: Tool chain for test case generation from UML models.

preserving coverage of model mutations. Furthermore, we present a small case study of a car alarm system (CAS) provided by Ford.

## 6.1. Testing Object-oriented Systems

In order to alleviate the modeling of object-oriented systems we have extended our action system models. We represent an action system $AS$ comprising a vector of variables $V$ of types $T$, initialized with values $I$, $n$ methods $M$, $m$ named actions $A$, $d$ action calls $C_i \in A$ composed via one of the operators $\texttt{op} =_{df} \Box \mid ; \mid //$, and a set $M_I$ of imported methods syntactically as follows:

$$
\begin{array}{lll}
AS_O & =_{df} & |[ \quad \textbf{var} \;\; V : T = I \\
& & \qquad \textbf{methods} \\
& & \qquad M_1, \ldots, M_n \\
& & \qquad \textbf{actions} \\
& & \qquad A_1, \ldots, A_m \\
& & \qquad \textbf{do}\; C_1 \;\texttt{op}\; C_2 \ldots \texttt{op}\; C_d \;\textbf{od} \\
& & ]| : M_I
\end{array}
\tag{6.1}
$$

Methods have a name, a body and may have a return value. Named actions are similar to methods but have no return value. The box-operator ("$\Box$") stands for non-deterministic, demonic choice. The operator ";" denotes sequential composition, and the "$//$" operator stands for prioritizing composition [145]. A prioritizing composition $A//B$ means that if $A$ is enabled it will be executed, otherwise action $B$ is executed. This operator can be rewritten into a non-deterministic choice, namely $A \Box \neg g(A) \rightarrow B$, where $g(A)$ denotes the enabledness guard of action $A$.

Methods and named actions are composed of atomic actions which are linked with the operators op. Atomic actions are guarded commands, (nondeterministic) assignments, method calls, *skip*, and *abort*. Given the definition of methods and actions the behavior of action executions is specified in the **do od** section. In contrast to conventional action systems where all actions are composed via nondeterministic choice we allow here arbitrary compositions with the operators op. This brings the advantage that one is able to specify the behavior of state machines without encoding the transition relation via an additional state variable and according conditions in action guards.

We assume that named actions may only be called from within the **do od** (that is, not from within named actions or methods), and that method calls must not be recursively nested. We also demand that each named action has the form of a guarded command. Relying on these assumptions, we are allowed to rewrite the action system into the classical form, see Definition 5.2, where

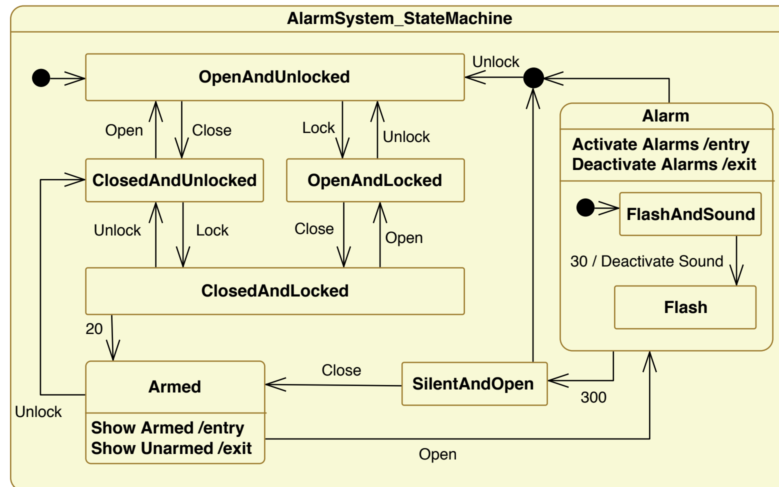



Figure 6.2.: Car Alarm System - state machine.

only the actions within the **do od** are left. This requires the elimination of method calls by replacing the calls with their method bodies (method inlining). Furthermore, referenced variables, which are external, have to be added to the import list.

In a first version of Ulysses the actions of a given model were interpreted. However, for larger models interpretation results in quite long test generation times. Thus, we implemented a compilation mode where purely discrete action systems are translated to plain Prolog clauses. The exploration of a compiled action system is at an exponential factor faster than the exploration in interpretation mode.

The development of specifications is easier when they are "executable" which is usually the case for formal models. Since our models are nondeterministic, execution means model animation. Ulysses features a model animator that provides a user the choice of the next events in the suspension automaton of a given (qualitative) action system. This enables the designer to try out what has been modeled so far and is an important means for model validation.

## 6.2. Car Alarm System

The UML testing model for the car alarm system (CAS) was created from the following list of requirements:

**R1: Arming.** The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.

**R2: Alarm.** The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

**R3: Deactivation.** The CAS can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

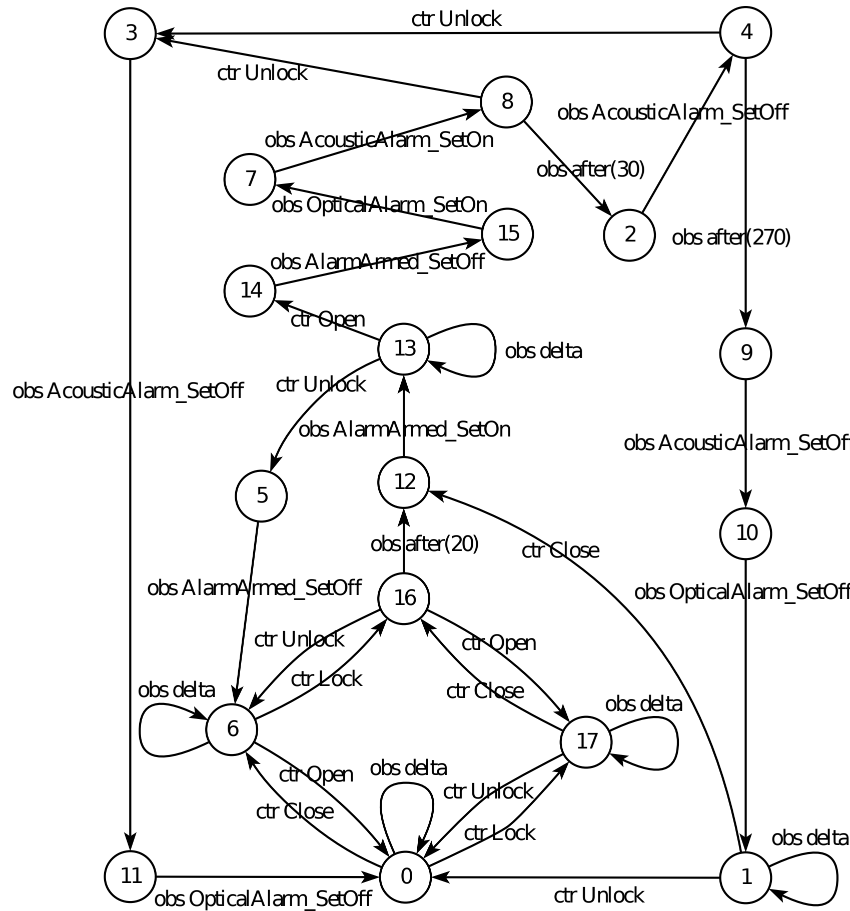



Figure 6.3.: Suspension automaton of the Car Alarm System.

When someone constructs a formal specification based on textual requirements, often conflicts or underspecified situations become apparent. One might think that the simplistic car alarm system is sufficiently described by these three textual requirements – the contrary is the case. What is left unspecified is the case of what happens when an alarm is ended by the five minutes timeout: does the system go back to armed directly, or does it need to wait for all doors to be closed again before returning to armed? For our model, we chose the latter option.

**State Machine.** Figure 6.2 shows the CAS state machine diagram. The events *Lock*, *Unlock*, *Close*, and *Open* are controllable while the events *AlarmArmed*, *AcousticAlarm*, and *OpticalAlarm* are observable by the tester.

From the state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car. Actions of closing, opening, locking, and unlocking are modeled by corresponding signals *Close*, *Open*, *Lock*, and *Unlock*. As specified in the first requirement, the alarm system is armed after 20 seconds in *ClosedAndLocked*. Upon entry of the *Armed* state, the model calls the method *AlarmArmed.SetOn*. Upon leaving the state, which can be done by either unlocking the car or opening a door, *AlarmArmed.SetOff* is called. Similar, when entering the

*Alarm* state, the optical and acoustic alarms are enabled. When leaving the alarm state, either via a timeout or via unlocking the car, both acoustic and optical alarm are turned off. When leaving the alarm state after a timeout, turning off the acoustic alarm after 30 seconds, as specified in the second requirement, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

Figure 6.3 depicts the suspension automaton of the CAS model. The "`ctr`" and "`obs`" prefixes denote controllable and observable events respectively. The visible behavior of the UML model is quite small, however the mapping from UML to action systems results in an overhead of internal actions which should not be underestimated. The CAS input model for Ulysses contains 750 lines.

**Mutations.** As we want to create test cases that cover particular fault models, we need to deliberately introduce 'bugs' in the specification model. In order to do that, we rely on different mutation operators. As an example, one mutation operator sets guards of transitions to false[1], while others remove entry actions, signal triggers, or change signal events. Applying these operators to our CAS specification model yields 76 mutants: 19 mutants with a transition guard set to false, 6 mutants with a missing entry action, 12 mutants with missing signal triggers, 3 with missing time triggers, and 36 with changed signal events.

**Example 6.1.** Let us consider a mutant which lacks the transition from *OpenAndUnlocked* to *OpenAndLocked*. This means that the system simply ignores the *Lock* event when being in the *OpenAndUnlocked* state instead of proceeding to *OpenAndLocked*.

The first two LTSs in Figure 6.4 show the partial suspension automata of the specification and the mutant. The mutant is not *ioco* to the specification because the subset inclusion of the observations after the trace $\langle Lock, Close \rangle$ does not hold, i.e., $out(Mutant\ after\ \langle Lock, Close \rangle) = \{\delta\} \not\subseteq out(Spec\ after\ \langle Lock, Close \rangle) = \{after(20)\}$. The product LTS (depicted on the right-hand side) may contain states with *pass* self-loops. They indicate that the specified behavior is left after an unspecified controllable event or ends after a not implemented observable event. Furthermore, *fail* states, i.e., states with *fail* self-loops, denote that an unspecified observable event has occurred. In this case, we are able to derive a controllable test case that is able to detect this unspecified behavior (cf. LTS on the right-hand side). □

## 6.3. Experimental Results

This section presents an evaluation of our tool chain where we apply eight different test case generation approaches to the car alarm system. We start our discussion with a summary of the different test case generation approaches before evaluating their fault detection capabilities.

[1] This may lead to a transition transforming into a self loop as the model will 'swallow' the trigger event

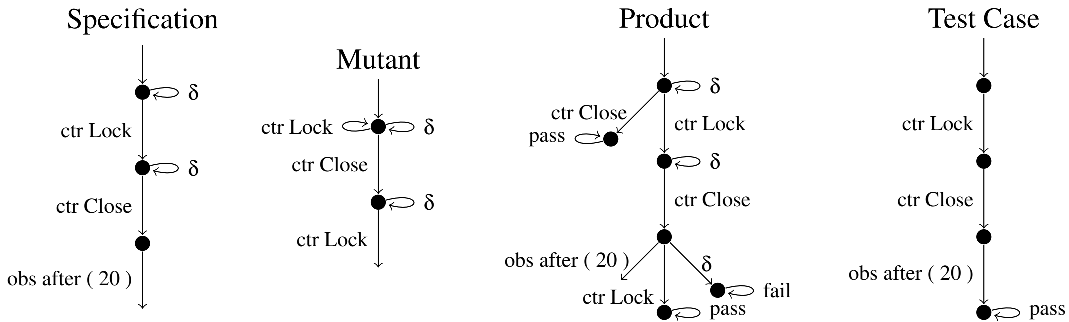



Figure 6.4.: The two LTSs on the left show parts of the suspension automata of the CAS specification and a mutant. The two LTSs on the right depict the resulting *ioco* product and a selected test case.

| | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 |
|---|---|---|---|---|---|---|---|---|
| Max. Depth | 10 | 14 | 23 | 23 | 23 | 150 (19) | 150 | 30 |
| Gen. TCs [#] | 16 210 | 302 | 504 | 129 | 63 | 11 | 3 | 9 |
| Duplicates [#] | 12 179 | 174 | 217 | 0 | 0 | 0 | 0 | 0 |
| Unique [#] | 3 469 | 110 | 269 | 123 | 59 | 11 | 3 | 9 |
| Gen. Time [min] | 188 | 91 | 23 | 70 | 23 | 10 | 0.25 | - |

Table 6.1.: Number of generated test cases

### 6.3.1. Test Case Generation

We compare eight different test case generation approaches called A1 to A8. Generally speaking, A1 to A6 are fault-based approaches, A7 is based on random testing, and approach A8 uses test cases that were manually designed. All fault-based approaches use the set of 76 faulty specifications as input to test case generation. Note that because the car alarm system is deterministic, all test cases are linear. While the test cases in approach A1 and A2 have been generated via straightforward path search the approaches A3–A6 employ the test selection Algorithm 5.2. In A6 we generate test cases randomly and due to Algorithm 5.2. Table 6.1 highlights the main differences between the approaches regarding generated test cases.

**Approach A1** Within the first approach, the product graph of the *ioco* check is first transformed into a tree structure with a maximum depth of 10 and a two-times maximum visit per state per trace. After tree unfolding the graph, we generate linear test cases for all paths that lead to an unsafe state in the tree. As can be seen in the table, this greedy test case generation strategy produces a very large number of tests. Also, approximately 75% of the generated tests are duplicates. From the non-duplicate test cases, we could exclude another 562 tests that are a complete subsequence of one of the other remaining test cases. Hence, about 3500 tests remain. Due to the vast amount of test cases generated by A1, we refrained from executing the tests on the CAS implementations.

**Approach A2** The second test case generation strategy directly works on the product graph and extracts only one arbitrary linear test per unsafe state. Notice that one mutant may still involve more than one unsafe state in the product graph. This time, we could identify 58% of duplicate test cases and another 22 were covered as a subsequence in other test cases. It has to be said that we allowed for tests with a maximum length of 14 in this approach, thus generation times are not directly comparable between A1 and A2. We verified that all unique test cases with a length of up to 10 that were produced by this approach were included in the set of tests generated by the first approach.

**Approach A3** The third approach is similar to the second one, except that the depth is theoretically unbounded since the test case generator applies algorithm 5.2. Therefore, the generation time is not comparable with approaches A1 and A2 (but with A4 to A7).

**Approach A4** The fourth approach builds on A3 but avoids creating test cases for unsafe states in a given product graph which are already covered by existing test cases. In order to check if a certain unsafe state is already covered by an existing test case we generate a test purposes *tp* for that unsafe state. Then the existing test cases are checked by computing the synchronous product with *tp*. This is similar to the synchronous product calculation in the tool *TGV*[98], however *TGV* computes it between test purpose and specification. A test case *tc* satisfies a test purpose if the product calculation prunes no transitions from *tc*, i.e. $tc \times tp = tc$.

The generated test purpose ensures that a given test case passes through the unsafe state to a pass state. Given an unsafe state $u$ we determine the set of states $L2U$ which lead to $u$, i.e. $L2U =_{df} \{s \in S \mid \exists \sigma \bullet s \stackrel{\sigma}{\Rightarrow} u\}$. The set of pass states associated with $u$ are $Pass =_{df} \{s \in S \mid \exists a \in L_U \bullet (u,a,s) \in T \wedge s \notin Fail\}$. The resulting test purpose $tp_u$ has the transition relation $T_u =_{df} \{(s_1,a,s_2) \in T \mid s_1, s_2 \in L2U \vee (s_1 = u \wedge s_2 \in Pass)\} \cup \{(s,*,s) \mid s \in Pass\}$. The remaining behavior after reaching a state $s \in Pass$ is not of interest, denoted by the $*$ label allowing any transitions to occur thereafter.

This approach yields one test case per unsafe state. Note however, that generated test cases still can be included in other test cases. This depends on the order of processed unsafe states. If we would start with deep unsafe states and proceed to shallow ones it is theoretically possible that no generated test case is included in another one.

While the total number of generated test cases decreases to 129 (from 504), the time used to generate them increases to 70 minutes (from 23). This is because we compute the check for every unsafe state and in the worst case for all test cases generated so far. In effect, A4 checks whether an existing test case already covers an unsafe state in the *ioco* product before creating a new test case. If an unsafe state (there may be several per mutated specification) is not covered, a new test case is emitted.

**Approach A5** Approach A5 also avoids creating duplicate test cases. However, A5 further tries to minimize the size of the generated test suite. Before creating test cases for a mutated specification, A5 first checks whether any of the previously created test cases is able to kill the mutated specification. Therefor, the synchronous parallel execution $\rceil|$ of the test case *tc* with

the mutated specification under test $m$ is applied. The following rules define the operational semantics for killing a mutated specification.

$$\frac{tc \xrightarrow{a} tc' \quad m \xrightarrow{a} m'}{tc \rceil | m \xrightarrow{a} tc' \rceil | m'} \ (1) \qquad \frac{a \in L_I \quad tc \xrightarrow{a} tc' \quad m \not\xrightarrow{a} m'}{tc \rceil | m \xrightarrow{a} tc' \rceil | m} \ (2) \qquad \frac{a \in L_U \quad tc \not\xrightarrow{a} tc' \quad m \xrightarrow{a} m'}{tc \rceil | m \xrightarrow{a} fail \rceil | m'} \ (3)$$

Rule (1) describes the case where $tc$ and $m$ synchronize on common events and proceed to their next state. Due to the input enabledness assumption the mutated specification has to accept all input events in every state. This is realized by making every state of $m$ input complete by adding self loops for the remaining input events, see Rule (2). Rule (3) states that the test case reaches a fail state when the mutant produces an output event which the test case cannot follow. Given these rules a mutant is killed when a fail state can be reached, i.e.

$$\exists \sigma \in (L \cup \delta)^* \bullet tc \rceil | m \overset{\sigma}{\Rightarrow} fail \rceil | m'.$$

The check is done without calculating the full *ioco* product between the specification and the mutant, which is the first difference to A4. A4 always calculates the full *ioco* product. If an existing test is able to kill the mutant, the test suite is considered strong enough and no new test cases are generated for the mutant. Notice that this is the second difference to A4 as A4 will only skip test case generation for covered unsafe states, while A5 does never generate any additional test case for a killed mutant. Due to this minimization, approach A5 is sensitive to the ordering of mutants. Notice that A5 starts with an empty test suite.

**Approach A6** Approach A6 is like A5 but instead of starting with an empty test suite, A6 uses one randomly generated test case to start with. In effect, A6 is a combination of random (A7) and fault based testing (A5). Within Table 6.1, the maximum depth not put in brackets is the depth of the randomly generated initial test case while the figure in brackets is the maximum depth of the additionally generated tests to cover all faulty specifications.

**Approach A7** This approach uses a random selection strategy in order to generate test cases. Put differently, A7 is not a fault-based test case generation approach and included for evaluation purposes only.

**Approach A8** Finally, we compare our test case generation approach based on UML with manually created test cases. More precisely, we generated 9 different test purposes by hand and let TGV [98] create test cases. 3 out of the 9 test cases check for observable timeouts (time-triggered transitions: 20, 30, 300 sec. delay). 4 test cases check the entry and exit actions of the states *Armed* and *Alarm*. One test case checks for the deactivation of the acoustic alarm after the timeout and one more complex test case has a depth of 30 transitions going once through the state *SilentAndOpen* to *Armed* before going to *Alarm* again and leaving after the acoustic alarm deactivation by an unlock event. Hence, each observable event is covered by at least one test case. During the creation of the test purposes, we relied on a printout of the UML state machine.

| | Mutants | Equiv. | Pairwise Equiv. | Different Faults |
|---|---|---|---|---|
| SetState | 6 | 0 | 1 | 5 |
| Close | 16 | 2 | 6 | 8 |
| Open | 16 | 2 | 6 | 8 |
| Lock | 12 | 2 | 4 | 6 |
| Unlock | 20 | 2 | 8 | 10 |
| Constr. | 2 | 0 | 1 | 1 |
| Total | 72 | 8 | 26 | 38 |

Table 6.2.: Injected faults into the CAS implementation.

### 6.3.2. Test Case Execution

We applied classical mutation analysis in order to evaluate the effectiveness of our different test suites. For this purpose, we have implemented the CAS in Java based on the state machine of Figure 6.2. In order to derive a set of faulty implementations, we used the $\mu$Java [121] tool: in total, $\mu$Java gave us 72 mutated implementations. After careful inspection, 8 of these mutated implementations were found to be equivalent to the original program, and another set of 26 mutants was found to be equivalent to other mutants - forming 26 equivalent pairs. Hence, a sum of 38 ($72 - 8 - 26$) different faulty implementations of the CAS remain. Further details are shown in Table 6.2. For each method, the table lists the total number of mutated implementations, the number of mutants that turned out to be equivalent to the original implementation, and the number of equivalent pairs of mutated implementations. The methods *Close*, *Open*, *Lock*, and *Unlock* are public ones and handle the equally named external events while *SetState* and the constructor (*Constr*) are internal methods. From the table, one can observe that the mutation on internal methods has a strong effect on the external behavior since there are no equivalent mutants for these methods.

In the following, we use the 38 unique faulty CAS implementations to evaluate the effectiveness of our generated test cases. Of course, all tests were validated on the non-mutated CAS implementation.

Table 6.3 gives an overview of the number of survived faulty implementations for each test case generation approach. As can be seen from the table, the approaches A3, A4, and A6 were able to reveal all faults. The table also shows that the minimization algorithm applied in approach A5 reduces the fault-detection capabilities. Despite random testing (A7) did not find all faulty implementations, it proved quite effective in our setting. It has to be noted, however, that we allowed for an appropriate depth of the random test cases. Summing up, A5 and A7 still have a fault detection rate of 97%. The reason for the bad performance of A2 is the fact that the two surviving faulty implementations need tests with a depth of more than 14 interactions to be revealed and A2's tests are restricted to a depth $\leq 14$.

The last column shows the results of running the manually designed tests (A8). Overall, 25 mutants were killed which results in a detection rate of 66%. Clearly, the figures show that in order to have a meaningful test suite, more (diverse) test cases have to be generated. Partly, this lack

| | A2 | A3 | A4 | A5 | A6 | A7 | A8 |
|---|---|---|---|---|---|---|---|
| SetState | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Close | 1 | 0 | 0 | 0 | 0 | 0 | 2 |
| Open | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| Lock | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| Unlock | 1 | 0 | 0 | 0 | 0 | 1 | 5 |
| Constr. | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 2 | 0 | 0 | 1 | 0 | 1 | 13 |
| Detection Rate [%] | 95 | 100 | 100 | 97 | 100 | 97 | 66 |
| Impl. Coverage [%] | 99 | 99 | 99 | 99 | 99 | 99 | 88 |

Table 6.3.: Overview of how many faulty SUTs could not be killed by the test cases generated with different approaches.

| | A3 | A4 | A5 |
|---|---|---|---|
| Gen. TCs [#] | 469 | 58 | 32 |
| Gen. Time [min] | 10:05 | 21:52 | 14:49 |

Table 6.4.: Number of generated test cases (hand-written OOAS model)

of diverse test cases is based on the deterministic test selection behavior of TGV: all TGV based tests have almost the same test sequence from *OpenAndUnlocked* to *ClosedAndLocked*, although alternative paths are possible.

Often, coverage metrics on the implementation's source code serve as a quality measure to describe the adequacy of a test suite. Therefore, we have measured the coverage on the implementation in terms of basic block coverage. The approaches A2 - A7 achieve a coverage of 99%, whereas A8 (TGV) only results in a basic block coverage of 88%. By comparing the last two rows of Table 6.3, it becomes obvious that a high code coverage does not automatically guarantee high fault detection rates, which we aim for.

In summing up, the results show that our approaches are powerful in covering the implementation's source code and more importantly in detecting faults. However, the depth of our conformance analysis is critical as too less depth results in missing test cases and, in this example, in undetected faults. The results also show that the 3500 test cases of the first approach were by far too many: for the given model mutations, one path per mutation is sufficient to detect all faults, provided this path is long enough. Finally, the combined approach (A6) proved to be a nice trade-off between generation time and effectiveness.

Beside using UML models we directly modeled the CAS in OOAS language. The transformation from UML to OOAS brings quite an overhead in lines of code: while the hand-written OOAS model contains 113 lines the transformed UML model, as already stated, results in 750 lines. This is mainly because UML is intended to specify systems on the implementation level rather than on an abstract requirements level. For instance, the whole event signaling mechanism of UML has to be transformed to OOAS constructs.

For a comparison with the UML-based approach we implemented a mutation tool which cre-

ated 442 mutated OOAS specifications. Table 6.4 shows the number of test cases with according generation times for approaches A3-A5. Each test suite of the three approaches was able to detect all faulty implementations. The results show that our selection strategies are able to efficiently minimize test suites for a large number of mutated specifications.

Chapter 7

# Conclusions

This chapter presents a summary of the previous chapters, draws some conclusions, and gives an outlook for future work.

## 7.1. Summary

After an introduction to the content of this theses we discussed its broader context of model-based testing. Next, Qualitative Reasoning and its application for modeling continuous systems was covered. We discussed how qualitative models can be simplified while preserving the behavior which is of interest for testing.

We covered the application of model-based testing of continuous systems using qualitative models. In order to decide the correctness of an implementation regarding a qualitative model the conformance relation *qrioconf* was introduced. All results were generated with the prototype tool *QRPathfinder* which is implemented in *Java1.6*. The tool employs the *InterProlog*[1] interface for connecting to the QR tool *Garp3* [44]. *QRPathfinder* loads a qualitative transition system which is obtained by simulating a *Garp3* model, generates test cases according to test purposes or coverage criteria, and provides a test adapter for executing test cases on Matlab/Simulink [13] models.

The experiments showed that our approach is able to detect certain kinds of faults in continuous systems. Qualitative models are well suited to specify a wider class of implementations than is possible by using parameterized ODEs. A further benefit of QR is that some system parameters may even be unknown. Numerical values become only relevant during test case execution for variables at the testing interface.

The good fault detection capability of qualitative test cases can be explained due to the transfer behavior of continuous systems. A fault in such a system either changes a transfer function, or the transitions between transfer functions. Depending on the number of system modes, more

[1] http://www.declarativa.com/interprolog/

or less different test cases are required to exercise the different behaviors. Usually, a faulty transfer function immediately propagates an error from the inputs to the outputs. Thus, a faulty implementation can be detected after applying only a few input samples.

While conducting the experiments it turned out that state spaces for even medium sized qualitative models grow quite fast. This is because *Garp3* supports no hiding of variables. Thus, uninteresting, internal behavior is enumerated as well. Also, simulation times and memory consumptions are an issue. Therefore, it would be beneficial to apply online testing where the qualitative transition system is only unfolded as required. However, the simulation engine has to be fast enough to follow the responses of an IUT. If such timing issues cannot be met, passive testing is another alternative. Here, recorded traces of the implementation are replayed on the qualitative model.

For specifying test models of hybrid systems conventional action systems [15], more precisely a hybrid version thereof [139], were extended to so-called *qualitative action systems (QAS)* [6]. Each action is associated with a name (label) and the execution of a labeled action system results in a labeled transition system (LTS). The LTS semantics enables the application of standard conformance relations like input-output conformance [156].

After this, we presented the automated input-output conformance verification between two given QAS and its application in mutation-based test case generation. The ioco relation is defined over traces rather than over states like simulation relations. This requires the determinization of LTSs which we carry out on-the-fly. However, determinization and the associated merging of states may lead to enabledness problems of actions. This occurs when in a certain state an action is not enabled for all its substates. We discussed this problem and proposed an approach how to resolve it.

We applied several mutation operators to the model of a two-tank system and showed the results of the conformance check between original and mutated model. The presented test case selection algorithm produces minimal and adaptive test cases by computing shortest traces which reveal the mutation. The obtained test cases are able to check the behavior of a control program as well as the outcomes after continuous changes.

In mutation-based test case generation, the question arises which mutant can be killed by test cases generated from other mutants and for which mutant new test cases have to be generated. The answer to this question allows to create a minimal test suite for a given number of mutants. However, the check to determine what test case is able to kill which mutants can get very costly. The asymptotic runtime is $O(m \cdot (n-1))$, where $m$ is the number of non-equivalent mutants and $n$ is the total number of generated test cases. Thus, we applied different strategies that reduce the size of a test suite on-the-fly during test case selection. A comparison of these strategies was conducted on a small industrial case study. The results showed that the number of test cases can be significantly reduced while maintaining a high fault detection rate.

All results for the mutation-based test case generation from (qualitative) action system models were produced with our tool called *Ulysses*. It was implemented in SICStus Prolog[2] and employs the QR tool *ASIM* [22] for solving qualitative differential equations. *Ulysses* applies an exhaustive search strategy for exploring the behavior of QAS models. The conformance checker searches the

[2] www.sics.se/sicstus

state space until a difference with respect to *ioco* between original and mutated model is found or a defined search depth is reached. Due to interleavings of parallel behavior and the enumeration of data types the size of state spaces can grow to large numbers.

## 7.2. Related Research

While there exist various approaches to abstract continuous behavior to finite state representations [32, 33, 119, 90, 152] most of them rely on concrete values. With the technique of Qualitative Reasoning we follow a purely symbolic approach which allows us to build generic models of continuous and hybrid systems.

Our approach of generating qualitative test cases for continuous systems is mostly related to the approach implemented in the tool *TGV* [98]. In contrast to *TGV*, where test purposes are formulated as automata, we specify test purposes with regular expressions over symbolic properties. As a second strategy we generate qualitative test cases due to coverage criteria. This is similar to the approach in [75] where a model checker is used to create test cases according to so-called trap properties. We have developed the conformance relation *qrioconf* which is based on the *ioco* relation by Tretmans [156]. To our best knowledge this is the first work which employs qualitative reasoning for testing continuous and hybrid systems.

In this thesis the part of testing hybrid systems is based on the concepts of qualitative reasoning [112] and action systems [15]. Most relevant to our research are different extensions to action systems that allow modeling of hybrid systems. The work in [19] presents *continuous action systems*. They are similar to conventional action systems except for the fact that continuous functions are used as values for variables (attributes). The implicit attribute *now* represents the current time and starts at zero. Our approach of testing hybrid systems is based on the framework of *hybrid action systems* [139]. The weakest precondition semantics of discrete as well as continuous actions enable us to define a labeled transition system (LTS) semantics of hybrid systems. We apply the conformance relation *ioco* for generating mutation-based test cases, similar to the work in [163]. The authors in [163] apply the ioco check between an original and a mutated specification in order to derive a test purpose which is then used by the tool TGV [98] for test case generation. In our work we directly extract adaptive test cases from the result of the ioco check.

This work is related to mutation-based test case generation. The authors of [49] present a first work on this topic. Aichernig and Jifeng [3] introduce a mutation-based testing approach in the Unifying Theories of Programming (UTP). Weiglhofer et al. [165] present the application of mutation testing to an industrial protocol specification. A survey on mutation testing can be found in [99].

Abstractions of hybrid systems are a common way to deal with inherent system complexity. The authors of [135] summarize results for property-preserving abstractions of hybrid systems. A combination of ideas taken from predicate abstraction and qualitative reasoning has been recently proposed in [151]: based on hybrid automata, the author presents a procedure for constructing sound abstractions for hybrid systems and also discusses which abstractions should be chosen. Abstract models of hybrid systems, besides being useful in automated verification, are also a

common way of specifying systems. This is the intended use of our qualitative action systems. Related work in qualitative modeling of hybrid systems can be found, e.g., in [148] where the authors present *Qualitative Charon*, a qualitative modeling language that is based on *Charon* [12] and qualitative reasoning [112].

The authors in [33] present a hybrid denotational semantics for hybrid systems. The presented approach provides a means to analyze hybrid systems by considering the continuous behavior only at certain points in time which are defined by the discrete system. For instance if the discrete system does not access sensor data from the environment it is not necessary to compute the state of the environment. The weakest-precondition semantics of our qualitative action systems is also denotational, but on the abstract qualitative level.

Reactis [136] is a common test generation tool for Simulink models. It produces test cases regarding certain coverage criteria on the Simulink model. Test cases can be used to ensure conformance of a source code implementation or to validate the behavior of the model itself. Reactis does not deal with mutation testing and works on the implementation level. It is designed for white-box testing, our tools target black-box testing.

The work in [68] deals with randomized test case generation for hybrid systems. The approach is based on the notation of hybrid automata and the idea is to explore the state space by building Rapidly Exploring Random Trees (RRTs) [115]. The RRT algorithm has been used in robotics for path planning by computing control signals for trajectories in high dimensional spaces. For testing, the RRT algorithm is used to find counter examples, i.e. input sequences that drive the system into states that are not in a defined specification set. The authors in [127] extend the random exploration technique with coverage information. Less explored regions are preferred in the exploration process. The usability and performance of RRTs depends on finding appropriate metrics in a system's state space. In random testing a large number of long test runs might be necessary in order to achieve a sufficient testing coverage. Our approach is more costly in the generation of test cases, however execution times are lower because of fewer, and more diverse test cases.

The selection of test cases for hybrid systems is addressed in [103]. The idea is that a test case with certain parameters has many test cases with slightly changed parameters in its neighborhood which all show the same qualitative behavior. The work deals with the generation of test cases based on coverage metrics. Similarly, in our work we apply qualitative reasoning as abstraction technique.

## 7.3. Future Work

As a next step the elaboration of online testing of continuous systems would be interesting. This approach eliminates the exploration of spurious behavior at all and reduces behavior branching only to the relevant choices of the current test run. Online testing promises to allow the handling of larger models.

For future extensions of *Ulysses* we identify three approaches for which an evaluation would be interesting:

- the integration of online testing into *Ulysses* to counter state space explosion,
- in order to reduce unnecessary interleavings a technique called *partial order reduction* can be applied,
- and the evaluation of a symbolic test case generation approach may lead to better computation times and lower memory consumptions. Symbolic test case generation could be further improved by employing counterexample guided abstraction refinement (CEGAR).

Writing formal specifications is similar to writing programs. The performance of analyzing a model mainly depends on its structure and the constructs used. Thus, a profiler for examining the bottlenecks in the exploration time of a model can be helpful.

The conformance relation *hioco* by van Osch [160] covers two parts. First, the discrete events have to obey the *ioco* relation. Secondly, trajectories of the implementation must be allowed by the specification. The implementation trajectories are filtered due to specified inputs. By replacing the trajectory inclusion check with our *qrioconf* relation we obtain a qualitative version of *hioco*. This allows to discover faults within continuous changes rather than only at their end. It would be interesting to compare such a relation with *ioco* in terms of their fault detection capabilities in hybrid systems.

# List of Acronyms

API . . . . . . . . . . . . . Application Programming Interface
ASO . . . . . . . . . . . . Association Shift Operator
CAS . . . . . . . . . . . . Car Alarm System
CLP . . . . . . . . . . . . Constraint Logic Programming
CSP . . . . . . . . . . . . Communicating Sequential Processes
CSP . . . . . . . . . . . . Constraint Satisfaction Problem
CTG . . . . . . . . . . . . Complete Test Graph
CUTE . . . . . . . . . . . Concolic Unit Testing Engine
DART . . . . . . . . . . . Directed Automated Random Testing
DFA . . . . . . . . . . . . Deterministic Finite Automaton
EFSM . . . . . . . . . . . Extended Finite State Machine
ENO . . . . . . . . . . . . Expression Negation Operator
ERO . . . . . . . . . . . . Event Replacement Operator
GPS . . . . . . . . . . . . Global Positioning System
GSM . . . . . . . . . . . . Global System for Mobile Communications
HAS . . . . . . . . . . . . Hybrid Action System
IDE . . . . . . . . . . . . . Integrated Development Environment
IEEE . . . . . . . . . . . . Institute of Electrical and Electronics Engineers
IOCO . . . . . . . . . . . Input Output Conformance
IOTS . . . . . . . . . . . . Input Output Transition System
IUT . . . . . . . . . . . . . Implementation Under Test
JML . . . . . . . . . . . . Java Modeling Language
LOTOS . . . . . . . . . Language Of Temporal Ordering Specification
LRO . . . . . . . . . . . . Logical Operator Replacement
LTS . . . . . . . . . . . . . Labeled Transition System
MCO . . . . . . . . . . . Missing Condition Operator
MOGENTES . . . . Model-based Generation of Tests for Dependable Embedded Systems
NIFS . . . . . . . . . . . . No Function In Structure
OCL . . . . . . . . . . . . Object Constraint Language

ODE ............ Ordinary Differential Equation
OOAS .......... Object Oriented Action System
QAS ............ Qualitative Action System
QDE ............ Qualitative Differential Equation
QR ............. Qualitative Reasoning
QTS ............ Qualitative Transition System
RAISE .......... Rigorous Approach to Industrial Software Engineering
RRO ............ Relational Replacement Operator
RRT ............ Rapidly exploring Random Tree
SCC ............ Strongly Connected Components
SEPIAS ......... Self Properties In Autonomous Systems
SMT ............ Satisfiability Modulo Theories
STG ............ Symbolic Test Generator
SUT ............ System Under Test
TDIOHS ........ Time Discrete Input Output Hybrid System
TIOTS .......... Timed Input Output Transition System
UML ........... Unified Modeling Language
UTP ............ Unified Theories of Programming
VDM ........... Vienna Development Method

# Bibliography

[1] J.-R. Abrial. *The B-book: assigning programs to meaningsp*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5. (Cited on pages 11, 15, and 16.)

[2] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, School of Information and Computer Science, Georgia Inst. of Technology, Atlanta, Ga., Sept. 1979. (Cited on page 14.)

[3] Bernhard K. Aichernig and He Jifeng. Mutation testing in UTP. *Form. Asp. Comput.*, 21:33–64, February 2009. ISSN 0934-5043. doi: 10.1007/s00165-008-0083-6. URL `http://portal.acm.org/citation.cfm?id=1501948.1501955`. (Cited on pages 52 and 115.)

[4] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol conformance testing a SIP registrar: An industrial application of formal methods. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 215–224, London, UK, 2007. IEEE. (Cited on page 1.)

[5] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects (FMCO)*, volume 6286 of *Lecture Notes in Computer Science*, pages 228–249. Springer, 2009. (Cited on pages 7 and 73.)

[6] Bernhard K. Aichernig, Harald Brandl, and Willibald Krenn. Qualitative action systems. In Karin Breitman and Ana Cavalcanti, editors, *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM)*, volume 5885 of *LNCS*, pages 206–225. Springer, 2009. (Cited on pages 7, 25, 29, 73, and 114.)

[7] Bernhard K. Aichernig, Harald Brandl, Willibald Krenn, and Rudolf Schlatte. Model-based test case generation techniques: A survey. Technical report, Institute for Software Technology, TU Graz, 2009. (Cited on page 9.)

[8] Bernhard K. Aichernig, Harald Brandl, and Franz Wotawa. Conformance testing of hybrid systems with qualitative reasoning models. *Electron. Notes Theor. Comput. Sci.*, 253(2): 53–69, 2009. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2009.09.051. (Cited on pages 7, 49, and 50.)

[9] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: A two-layered interpretation for testing. In *UML&FM*, ACM Software Engineering Notes (SEN), 2010. in press. (Cited on pages 7 and 101.)

[10] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient mutation killers in action. In *Proceedings of the fourth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2011. in press. (Cited on pages 7, 73, and 101.)

[11] Rajeev Alur, Thomas Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin / Heidelberg, 1998. URL `http://dx.doi.org/10.1007/BFb0055622`. 10.1007/BFb0055622. (Cited on page 20.)

[12] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in charon. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 6–19, London, UK, 2000. Springer-Verlag. ISBN 3-540-67259-1. (Cited on pages 22 and 116.)

[13] Anne Angermann, Michael Beuschel, and Martin Rau. *Matlab - Simulink - Stateflow. Grundlagen, Toolboxen, Beispiele*. Oldenbourg, 5., aktualisierte edition, 2007. ISBN 3486582720. (Cited on pages 6, 23, 51, 69, and 113.)

[14] R. J. R. Back and F. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, 1988. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/48022.48023. (Cited on page 22.)

[15] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, Montreal, Quebec, Canada, 1983. ACM. (Cited on pages 6, 73, 114, and 115.)

[16] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991. (Cited on page 73.)

[17] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In B Jonsson and J. Parrow, editors, *CONCUR-94:Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384, Uppsala, Sweden, Aug 1994. Springer-Verlag. (Cited on page 73.)

[18] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus, a Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. (Cited on pages 15, 16, 84, and 85.)

[19] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Continuous action systems as a model for hybrid systems. *Nordic Journal of Computing*, 8(1):2–21, 2001. (Cited on pages 73 and 115.)

[20] Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test automation for hybrid systems. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, pages 14–21, New York, NY, USA, 2006. ACM. ISBN 1-59593-584-3. doi: http://doi.acm.org/10.1145/1188895.1188902. (Cited on page 23.)

[21] Elinor Bakker. Qualitative models of population dynamics. Master's thesis, University of Amsterdam, 2006. (Cited on page 45.)

[22] Aleksander Bandelj, Ivan Bratko, and Dorian Šuc. Qualitative simulation with CLP. In *In Proc. of 16th International Workshop on Qualitative Reasoning (QR'02*, 2002. (Cited on pages 26, 89, 91, and 114.)

[23] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005. (Cited on page 15.)

[24] A. F. E. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer Verlag, 2005. ISBN 3 540 26278 4. (Cited on page 11.)

[25] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010. URL `http://dx.doi.org/10.1007/978-3-642-12002-2_21`. (Cited on page 13.)

[26] Gilles Bernot. Testing against formal specifications: A theoretical view. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119, London, UK, 1991. Springer-Verlag. ISBN 3-540-53981-6. (Cited on page 15.)

[27] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003. (Cited on page 14.)

[28] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation Operators for Specifications. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on pages 14 and 98.)

[29] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 125–139, 2004. (Cited on pages 13 and 22.)

[30] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction, LNCS 1422*, pages 68–95. Springer, 1998. (Cited on page 101.)

[31] Cari Borrás. Overexposure of Radiation Therapy Patients in Panama: Problem Recognition and Follow-Up Measures. *Rev Panam Salud Publica*, 20(2/3):173–187, 2006. URL `http://journal.paho.org/uploads/1162234952.pdf`. (Cited on page 3.)

[32] Olivier Bouissou and Matthieu Martel. Abstract interpretation of the physical inputs of embedded programs. In Francesco Logozzo, Doron Peled, and Lenore Zuck, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin / Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-78163-9_8`. (Cited on pages 2, 23, and 115.)

[33] Olivier Bouissou and Matthieu Martel. A hybrid denotational semantics for hybrid systems. In Sophia Drossopoulou, editor, *ESOP'08/ETAPS'08: Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 63–77, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78738-0, 978-3-540-78738-9. (Cited on pages 2, 23, 115, and 116.)

[34] P. Bourque and R. Dupuis. Guide to the software engineering body of knowledge 2004 version. *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*, 2004. (Cited on page 9.)

[35] A. Bouwer, J. Liem, and B. Bredeweg. *User Manual for Single-User Version of QR Workbench*. Naturnet-Redime, STREP project co-funded by the European Commission within the Sixth Framework Programme (2002-2006), 2005. Project no. 004074. Project deliverable D4.2.1. (Cited on page 45.)

[36] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In Anders Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 298–302. Springer Berlin / Heidelberg, 1998. URL `http://dx.doi.org/10.1007/BFb0055357`. 10.1007/BFb0055357. (Cited on page 22.)

[37] Harald Brandl. Testing of hybrid systems using qualitative models. volume Proceedings of Formal Methods 2009 Doctoral Symposium, pages 46–52, 11 2009. URL `http://www.win.tue.nl/~mousavi/fm09ds.pdf`. (Cited on pages 7 and 49.)

[38] Harald Brandl and Franz Wotawa. Test case generation from QR models. In Ngoc Nguyen, Leszek Borzemski, Adam Grzech, and Moonis Ali, editors, *New Frontiers in Applied Artificial Intelligence*, volume 5027 of *Lecture Notes in Computer Science*, pages 235–244. Springer Berlin / Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-69052-8_25. (Cited on pages 7 and 49.)

[39] Harald Brandl, Gordon Fraser, and Franz Wotawa. A report on QR-based testing. In *22nd International Workshop on Qualitative Reasoning*, pages 1–9, 2008. URL www.cs.colorado.edu/~lizb/qr08/papers/Brandl.pdf. (Cited on pages 7, 49, and 52.)

[40] Harald Brandl, Gordon Fraser, and Franz Wotawa. QR-model based testing. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 17–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-030-2. doi: http://doi.acm.org/10.1145/1370042.1370046. (Cited on pages 7, 49, and 52.)

[41] Harald Brandl, Gordon Fraser, and Franz Wotawa. Coverage-based testing using qualitative reasoning models. In *SEKE*, pages 393–398, 2008. (Cited on pages 7 and 49.)

[42] Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. Automated conformance verification of hybrid systems. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 3–12, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4131-0. doi: http://dx.doi.org/10.1109/QSIC.2010.53. URL http://dx.doi.org/10.1109/QSIC.2010.53. (Cited on pages 7, 14, and 73.)

[43] B. Bredeweg, J. Liem, A. Bouwer, and P Salles. *Curriculum for learning about QR modelling*. Naturnet-Redime, STREP project co-funded by the European Commission within the Sixth Framework Programme (2002-2006), 2005. Project no. 004074. Project deliverable D6.9.1. (Cited on page 45.)

[44] Bert Bredeweg, Anders Bouwer, Jelmer Jellema, Dirk Bertels, Floris Floris Linnebank, and Jochem Liem. Garp3 - a new workbench for qualitative reasoning and modelling. In *Proceedings of 20th International Workshop on Qualitative Reasoning (QR-06)*, pages 21–28, Hannover, New Hampshire, USA, 2006. (Cited on pages 8, 26, 45, 49, and 113.)

[45] Ed Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 402–416. North-Holland, 1988. (Cited on page 18.)

[46] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin / Heidelberg, 2001. URL http://dx.doi.org/10.1007/3-540-45510-8_9. (Cited on page 11.)

[47] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. Lotos specifications, their implementations and their tests. In Richard Jerry Linn and M. Ümit Uyar, editors, *Conformance testing methodologies and architectures for OSI protocols*, pages 468–479, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press. ISBN 0-8186-5352-3. (Cited on page 18.)

[48] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-26278-4. (Cited on page 11.)

[49] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Comput. Lang.*, 10:63–73, January 1985. ISSN 0096-0551. doi: 10.1016/0096-0551(85)90011-6. URL `http://portal.acm.org/citation.cfm?id=3806.3811`. (Cited on pages 14 and 115.)

[50] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970. (Cited on page 3.)

[51] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006. (Cited on page 13.)

[52] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time simulink to lustre. In *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 84–99. Springer Berlin / Heidelberg, 2003. URL `http://dx.doi.org/10.1007/978-3-540-45212-6_7`. (Cited on page 23.)

[53] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in CSP. In Michael Butler, Michael Hinchey, and María Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer Berlin / Heidelberg, 2007. URL `http://dx.doi.org/10.1007/978-3-540-76650-6_10`. (Cited on page 17.)

[54] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 86–91, New York, NY, USA, 1993. ACM. ISBN 0-89791-577-1. doi: http://doi.acm.org/10.1145/157485.164585. URL `http://doi.acm.org/10.1145/157485.164585`. (Cited on page 13.)

[55] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. URL `http://dx.doi.org/10.1007/3-540-46002-0_34`. (Cited on page 13.)

[56] CNN. CNN transcripts, February 2007. URL `http://transcripts.cnn.com/TRANSCRIPTS/0702/24/tww.01.html`. (Cited on page 4.)

[57] Enrico Coiera. The qualitative representation of physical systems. *The Knowledge Engineering Review*, 7:55–77, 1992. (Cited on page 25.)

[58] Thao Dang and Tarik Nahhal. Using disparity to enhance test generation for hybrid systems. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes*

*in Computer Science*, pages 54–69. Springer Berlin / Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-68524-1_6. (Cited on page 24.)

[59] Conrado Daws, Alfredo Olivero, and Sergio Yovine. Verifying ET-LOTOS programmes with KRONOS. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 227–242, London, UK, UK, 1995. Chapman & Hall, Ltd. ISBN 0-412-64450-9. URL http://portal.acm.org/citation.cfm?id=646213.681510. (Cited on page 22.)

[60] Luca de Alfaro. Game models for open systems. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 269–289, 2003. (Cited on page 20.)

[61] Hidde De Jong. Qualitative simulation and related approaches for the analysis of dynamic systems. *Knowl. Eng. Rev.*, 19(2):93–132, 2004. ISSN 0269-8889. doi: http://dx.doi.org/10.1017/S0269888904000177. (Cited on page 39.)

[62] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11:34–41, April 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136. URL http://portal.acm.org/citation.cfm?id=1300736.1301357. (Cited on pages 6 and 14.)

[63] W. DeRoever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0521641705. (Cited on pages 15 and 83.)

[64] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990. ISBN 0-387-96957-8. (Cited on page 16.)

[65] David Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin / Heidelberg, 1990. URL http://dx.doi.org/10.1007/3-540-52148-8_17. (Cited on page 22.)

[66] Mark Dowson. The ARIANE 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997. ISSN 0163-5948. URL http://portal.acm.org/citation.cfm?id=251992. (Cited on page 4.)

[67] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006. (Cited on page 12.)

[68] J.M. Esposito. Randomized test case generation for hybrid systems: metric selection. *System Theory, 2004. Proceedings of the Thirty-Sixth Southeastern Symposium on System Theory*, pages 236–240, 2004. doi: 10.1109/SSST.2004.1295655. (Cited on pages 24 and 116.)

[69] Adam Farquhar, Benjamin Kuipers, Jeff Rickel, David Throop, and The Qualitative Reasoning Group. QSIM: The program and its use, July 1993. (Cited on page 32.)

[70] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967. URL http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf. (Cited on page 13.)

[71] Kenneth D. Forbus. Qualitative process theory. *Artif. Intell.*, 24:85–168, December 1984. ISSN 0004-3702. doi: 10.1016/0004-3702(84)90038-9. URL http://portal.acm.org/citation.cfm?id=2719.2721. (Cited on pages 25, 26, and 45.)

[72] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A symbolic framework for model-based testing. In *1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 40–54. Springer, 2006. (Cited on page 19.)

[73] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19:215–261, September 2009. ISSN 0960-0833. doi: 10.1002/stvr.v19:3. URL http://portal.acm.org/citation.cfm?id=1600258.1600261. (Cited on page 11.)

[74] Peter Fritzson. Modelica - a language for equation-based physical modeling and high performance simulation. In *Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, PARA '98, pages 149–160, London, UK, 1998. Springer-Verlag. ISBN 3-540-65414-3. URL http://portal.acm.org/citation.cfm?id=645781.666514. (Cited on page 22.)

[75] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin / Heidelberg, 1999. URL http://dx.doi.org/10.1007/3-540-48166-4_10. (Cited on page 115.)

[76] Marie-Claude Gaudel. Testing can be formal too. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, May 1995. (Cited on pages 9 and 12.)

[77] Marie-Claude Gaudel and Pascale Le Gall. Testing data types implementations from algebraic specifications. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing 2008*, volume 4949 of *LNCS*, pages 209–239, 2008. (Cited on page 11.)

[78] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal test-case generation for UML statecharts. In *ICECCS '04: Proc. of the 9th IEEE Int. Conf. on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age*, pages 75–84. IEEE Computer Society, 2004. (Cited on page 101.)

[79] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *SIGPLAN Not.*, 40:213–223, June 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1064978.1065036. URL http://doi.acm.org/10.1145/1064978.1065036. (Cited on page 12.)

[80] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability (STVR)*, 2010. (Cited on page 1.)

[81] Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen, and Rudolf Schlatte. Dynamic symbolic execution for testing distributed objects. In Catherine Dubois, editor, *Proceedings of the 3rd International Conference on Tests and Proofs*, volume 5668 of *Lecture Notes in Computer Science*, pages 105–120, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02948-6. doi: http://dx.doi.org/10.1007/978-3-642-02949-3_9. URL `http://dx.doi.org/10.1007/978-3-642-02949-3_9`. (Cited on page 13.)

[82] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. URL `citeseer.ist.psu.edu/halbwachs91synchronous.html`. (Cited on page 23.)

[83] T. A. Henzinger. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7463-6. (Cited on pages xiii, 2, 22, and 73.)

[84] Thomas Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond HyTech : Hybrid Systems Analysis Using Interval Numerical Methods. In Nancy Lynch and Bruce Krogh, editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 130–144. Springer Berlin / Heidelberg, 2000. URL `http://dx.doi.org/10.1007/3-540-46430-1_14`. (Cited on page 23.)

[85] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 373–382, New York, NY, USA, 1995. ACM. ISBN 0-89791-718-9. doi: http://doi.acm.org/10.1145/225058.225162. (Cited on page 23.)

[86] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *STTT*, 1(1-2):110–122, 1997. (Cited on page 23.)

[87] Anders Hessel and Paul Pettersson. A test case generation algorithm for real-time systems. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 268–273, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2207-6. doi: http://dx.doi.org/10.1109/QSIC.2004.5. (Cited on page 22.)

[88] Anders Hessel and Paul Pettersson. Model-based testing of a wap gateway: An industrial case-study. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 116–131, 2006. (Cited on page 22.)

[89] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer Berlin / Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-78917-8_3`. (Cited on pages 13, 19, 21, and 22.)

[90] Timothy J. Hickey and David K. Wittenberg. Rigorous modeling of hybrid systems using interval arithmetic constraints. In Rajeev Alur and George J. Pappas, editors, *Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 139–142. Springer Berlin / Heidelberg, 2004. URL `http://dx.doi.org/10.1007/978-3-540-24743-2_27`. (Cited on pages 2, 23, and 115.)

[91] Robert Hierons, Mercedes Merayo, and Manuel Núñez. Implementation relations for the distributed test architecture. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin / Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-68524-1_15`. (Cited on page 19.)

[92] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009. (Cited on page 11.)

[93] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985. (Cited on page 16.)

[94] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998. (Cited on page 16.)

[95] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, December 1990. (Cited on pages 9 and 10.)

[96] IEEE. IEEE standard for information technology - requirements and guidelines for test methods specifications and test method implementations for measuring conformance to POSIX(R) standards. *IEEE Std 2003-1997*, pages –, 2 Sep 1998. (Cited on page 12.)

[97] ISO. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989. (Cited on pages 13, 17, and 51.)

[98] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, August 2005. (Cited on pages 13, 21, 51, 53, 107, 108, and 115.)

[99] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62. (Cited on pages 14 and 115.)

[100] Einar B. Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society. doi: http://dx.doi.org/10.1109/SEFM.2004.6. URL `http://dx.doi.org/10.1109/SEFM.2004.6`. (Cited on page 13.)

[101] Cliff B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall, second edition, 1990. (Cited on pages 15 and 16.)

[102] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0262032708. (Cited on pages 9 and 53.)

[103] A. Julius, Georgios Fainekos, Madhukar Anand, Insup Lee, and George Pappas. Robust test generation and coverage for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo, editors, *Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 329–342. Springer Berlin / Heidelberg, 2007. URL `http://dx.doi.org/10.1007/978-3-540-71493-4_27`. (Cited on pages 23 and 116.)

[104] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin / Heidelberg, 1999. URL `http://dx.doi.org/10.1007/3-540-48683-6_12`. (Cited on page 54.)

[105] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. When BDDs fail: Conformance testing with symbolic execution and SMT solving. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 479–488, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3990-4. doi: http://dx.doi.org/10.1109/ICST.2010.48. URL `http://dx.doi.org/10.1109/ICST.2010.48`. (Cited on page 12.)

[106] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using uml statechart diagrams. In *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, SAICSIT '03, pages 296–300. South African Institute for Computer Scientists and Information Technologists, 2003. ISBN 1-58113-774-5. URL `http://portal.acm.org/citation.cfm?id=954014.954046`. (Cited on page 101.)

[107] Anne Keuneke and Dean Allemang. Exploring the no-function-in-structure principle. *J. Exp. Theor. Artif. Intell.*, 1:79–89, January 1990. ISSN 0952-813X. doi: 10.1080/09528138908953694. URL `http://portal.acm.org/citation.cfm?id=104943.104948`. (Cited on page 25.)

[108] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`. (Cited on page 12.)

[109] Johan De Kleer and John Seely Brown. A qualitative physics based on confluences. *Artif. Intell.*, 24(1-3):7–83, 1984. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(84)90037-7. (Cited on page 25.)

[110] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping UML to labeled transition systems for test-case generation – a translation via object-oriented action systems. In *Proc. of Formal Methods for Components and Objects (FMCO) 2009*, volume 6286 of *Lecture Notes in Computer Science*. Springer, 2010. (Cited on page 101.)

[111] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34:238–304, June 2009. ISSN 0925-9856. doi: 10.1007/s10703-009-0065-1. URL `http://portal.acm.org/citation.cfm?id=1541661.1541692`. (Cited on page 19.)

[112] Benjamin Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994. (Cited on pages 23, 25, 26, 27, 28, 29, 32, 34, 35, 36, 37, 38, 115, and 116.)

[113] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal: Status and future work. In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/326> [date of citation: 2005-01-01]. (Cited on page 22.)

[114] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne 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, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4. doi: http://doi.acm.org/10.1145/1086228.1086283. (Cited on page 22.)

[115] Steven M. LaValle and James J. Kuffner Jr. Randomized kinodynamic planning. *I. J. Robotic Res.*, 20(5):378–400, 2001. (Cited on pages 24 and 116.)

[116] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. URL `citeseer.ist.psu.edu/leavens99jml.html`. (Cited on page 15.)

[117] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 127–134, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://portal.acm.org/citation.cfm?id=872023.872569`. (Cited on page 52.)

[118] David Lee and Mihakus Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996. (Cited on page 11.)

[119] Marie-Anne Lefebvre and Hervé Guéguen. Hybrid abstractions of affine systems. *Nonlinear Analysis*, 65(6):1150 – 1167, 2006. ISSN 0362-546X. doi: DOI:10.1016/j.na.2005.12.016. URL `http://www.sciencedirect.com/science/article/B6V0Y-4JFGF71-2/2/9f7f4ee86fb16bf7f463a65950bc6feb`. Hybrid Systems and Applications (5). (Cited on pages 2, 23, and 115.)

[120] Nancy G. Leveson. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, 41:564–575, 2004. (Cited on page 4.)

[121] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: a mutation system for Java. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 827–830, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134425. (Cited on page 109.)

[122] K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. PhD thesis, Eindhoven University of Technology, 2006. (Cited on page 22.)

[123] Yuri V. Matiyasevich. *Hilbert's tenth problem*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-13295-8. (Cited on page 40.)

[124] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, 1997. ISBN 0-13-629155-4. (Cited on pages 13 and 15.)

[125] Carroll C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, 1990. (Cited on page 15.)

[126] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122. (Cited on page 3.)

[127] Tarik Nahhal and Thao Dang. Test coverage for continuous and hybrid systems. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 449–462. Springer Berlin / Heidelberg, 2007. URL `http://dx.doi.org/10.1007/978-3-540-73368-3_47`. (Cited on pages 24 and 116.)

[128] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7:165–192, 1997. (Cited on page 14.)

[129] A.J. Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. *Computer Assurance, 1996. COMPASS '96, 'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 224 –236, jun. 1996. doi: 10.1109/CMPASS.1996.507890. (Cited on page 14.)

[130] Alexandre Petrenko and Nina Yevtushenko. Queued testing of transition systems with inputs and outputs. In Rob Hierons and Thierry Jéron, editors, *Proceedings of the Workshop on Formal Approaches to Testing Of Software (Fates'02), A Satellite Workshop of Concur'02*, pages 79–94, Brno, Czech Republic, August 2002. (Cited on page 20.)

[131] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9103-8. (Cited on pages 23 and 73.)

[132] André Platzer and Jan-David Quesel. Keymaera: A hybrid theorem prover for hybrid systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer Berlin, Heidelberg, 2008. (Cited on page 23.)

[133] André Platzer and Jan-David Quesel. Logical verification and systematic parametric analysis in train control. In Magnus Egerstedt and Bud Mishra, editors, *Hybrid Systems:*

*Computation and Control*, volume 4981 of *Lecture Notes in Computer Science*, pages 646–649. Springer Berlin / Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-78929-1_55. (Cited on page 23.)

[134] The RAISE Language Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall, 1995. (Cited on pages 15 and 16.)

[135] Alur Rajeev, Thomas A. Henzinger, Gerardo Lafferriere, and George J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–983, 2000. (Cited on page 115.)

[136] Reactive Systems. Model-based testing and validation with Reactis. Technical report, Reactive Systems, Inc., 2003. (Cited on page 116.)

[137] Tomas Rokicki and Chris Myers. Automatic verification of timed circuits. In David Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 468–480. Springer Berlin / Heidelberg, 1994. URL http://dx.doi.org/10.1007/3-540-58179-0_76. (Cited on page 22.)

[138] Mauno Ronkko and Anders P. Ravn. Switches and jumps in hybrid action systems. Technical report, 1997. (Cited on page 78.)

[139] Mauno Rönkkö, Anders P. Ravn, and Kaisa Sere. Hybrid action systems. *Theor. Comput. Sci.*, 290(1):937–973, 2003. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(02)00547-9. (Cited on pages 6, 22, 73, 75, 78, 79, 84, 114, and 115.)

[140] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0136744095. (Cited on page 17.)

[141] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628. (Cited on page 11.)

[142] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002. ISBN 0137903952. (Cited on page 36.)

[143] Mauno Rönkkö and Kaisa Sere. Refinement and continuous behaviour. In Frits Vaandrager and Jan van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 223–237. Springer Berlin / Heidelberg, 1999. URL http://dx.doi.org/10.1007/3-540-48983-5_21. (Cited on page 83.)

[144] A. C. Cem Say and H. Levent Akin. Sound and complete qualitative simulation is impossible. *Artif. Intell.*, 149(2):251–266, 2003. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(03)00077-8. (Cited on page 40.)

[145] Emil Sekerinski and Kaisa Sere. A theory of prioritizing composition. Technical report, 1996. (Cited on page 102.)

[146] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In Thomas Ball and Robert Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer Berlin / Heidelberg, 2006. URL `http://dx.doi.org/10.1007/11817963_38`. (Cited on page 12.)

[147] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1595930140. doi: 10.1145/1081706.1081750. URL `http://portal.acm.org/citation.cfm?id=1081750`. (Cited on page 12.)

[148] Oleg Sokolsky and Hyoung Seok Hong. Qualitative modeling of hybrid systems. In *IN PROC. OF THE MONTREAL WORKSHOP*, 2001. (Cited on page 116.)

[149] Ian Stewart and David Tall. *The Foundations of Mathematics*. Oxford University Press, 1977. (Cited on page 43.)

[150] Nikolai Tillmann and Jonathan de Halleux. Pex–White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin / Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-79124-9_10`. (Cited on page 13.)

[151] Ashish Tiwari. Abstractions for hybrid systems. *Form. Methods Syst. Des.*, 32:57–83, February 2008. ISSN 0925-9856. doi: 10.1007/s10703-007-0044-3. URL `http://portal.acm.org/citation.cfm?id=1331427.1331462`. (Cited on page 115.)

[152] Ashish Tiwari and Gaurav Khanna. Series of abstractions for hybrid automata. In Claire Tomlin and Mark Greenstreet, editors, *Hybrid Systems: Computation and Control*, volume 2289 of *Lecture Notes in Computer Science*, pages 425–438. Springer Berlin, Heidelberg, 2002. URL `http://dx.doi.org/10.1007/3-540-45873-5_36`. (Cited on pages 2, 23, and 115.)

[153] C. Tomlin, G.J. Pappas, and S. Sastry. Conflict resolution for air traffic management: a study in multiagent hybrid systems. *Automatic Control, IEEE Transactions on*, 43(4):509 –521, April 1998. ISSN 0018-9286. doi: 10.1109/9.664154. (Cited on page 23.)

[154] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co. ISBN 0-444-81697-6. URL `http://portal.acm.org/citation.cfm?id=648128.747730`. (Cited on page 15.)

[155] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. URL `http://dx.doi.org/10.1007/3-540-61042-1_42`. (Cited on pages 6 and 18.)

[156] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996. (Cited on pages 18, 19, 49, 50, 95, 114, and 115.)

[157] Jan Tretmans. Model based testing with labelled transition systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. (Cited on pages 11, 13, 19, 55, and 56.)

[158] Jan Tretmans and Ed Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 13–25, Nurnburg, Germany, 2003. (Cited on page 13.)

[159] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. (Cited on page 11.)

[160] Michiel van Osch. Hybrid input-output conformance and test generation. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 70–84. Springer Berlin / Heidelberg, 2006. URL `http://dx.doi.org/10.1007/11940197_5`. (Cited on pages 19, 21, and 117.)

[161] Margus Veanes and Nikolaj Bjørner. Input-output model programs. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 322–335. Springer Berlin / Heidelberg, 2009. URL `http://dx.doi.org/10.1007/978-3-642-03466-4_21`. (Cited on page 20.)

[162] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin / Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-78917-8_2`. (Cited on pages 13 and 20.)

[163] Martin Weiglhofer and Franz Wotawa. "On the fly" input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 286–291, Anaheim, CA, USA, 2008. ACTA Press. ISBN 978-0-88986-716-1. URL `http://portal.acm.org/citation.cfm?id=1722603.1722655`. (Cited on pages 89 and 115.)

[164] Martin Weiglhofer and Franz Wotawa. Asynchronous input-output conformance testing. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*, pages 154–159, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3726-9. doi: 10.1109/COMPSAC.2009.194. URL `http://portal.acm.org/citation.cfm?id=1632705.1632998`. (Cited on page 93.)

[165] Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. Fault-based conformance testing in practice. *Int. J. Software and Informatics*, 3(2-3):375–411, 2009. (Cited on page 115.)

[166] W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent catastrophic accidents: Investigating how software was responsible. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '10, pages 14–22, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4086-3. doi: http://dx.doi.org/10.1109/SSIRI.2010.38. URL http://dx.doi.org/10.1109/SSIRI.2010.38. (Cited on page 4.)

[167] Franz Wotawa. Generating test-cases from qualitative knowledge – preliminary report. In *Proceedings of the 21st Annual Workshop on Qualitative Reasoning*, Aberystwyth, U.K., June 2007. (Cited on page 49.)