Dissertation

Model-Based Mutation Testing with Constraint and SMT Solvers

Elisabeth Jöbstl[1]

Institute for Software Technology (IST)
Graz University of Technology
A-8010 Graz, Austria

Supervisor/First reviewer: A.o. Univ.-Prof. DI Dr. Bernhard K. Aichernig
Second reviewer: Prof. Robert Hierons, Ph.D.

Graz, April 2014

[1] E-mail: joebstl@ist.tugraz.at

Dissertation

Modell-basiertes Mutationstesten mit Constraint- und SMT-Solvern

Elisabeth Jöbstl[1]

Institut für Softwaretechnologie (IST)
Technische Universität Graz
A-8010 Graz

Betreuer/1. Gutachter:	A.o. Univ.-Prof. DI Dr. Bernhard K. Aichernig
2. Gutachter:	Prof. Robert Hierons, Ph.D.

Graz, April 2014

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: joebstl@ist.tugraz.at

Abstract

Today, software is omnipresent in our everyday life. Computers and embedded systems are used naturally in telephones, cars, railway signalling systems, etc. For some of these applications, a malfunctioning is inconvenient – for others life-threatening. Hence, a thorough verification and validation of software systems is essential. In practice, it is mostly conducted via software testing.

In the last decade, model-based testing has been identified as a promising approach to software testing as it allows for automated test case generation. Instead of manually designing many individual test cases, the test engineer creates a formal model of the system under test and uses tools to automatically derive test cases. A crucial matter in model-based testing is the choice of the test criterion specifying which test cases shall be generated. In this work, we follow a fault-centred approach. The original test model is syntactically altered to produce a set of mutated models. We perform a conformance check between the original and the mutated models. In case of non-conformance, we automatically generate a test case that reveals the non-conforming behaviour of the mutant. When executed on the system under test, these test cases will detect whether one of the faulty models has been implemented instead of the correct, original model. Hence, the generated test suite covers all of the failures caused by the model mutation operators and has a high chance of covering many additional similar failures.

As a modelling formalism, we rely on action systems, which are well-suited to model reactive and non-deterministic systems. A main factor influencing the quality and performance of model-based mutation testing concerns the underlying conformance check, which is the main focus of this thesis. We designed and implemented an efficient test case generator for action systems based on two conformance relations: refinement and Input-Output Conformance (ioco).

For refinement checking, we defined a predicative semantics for action systems that is close to a constraint satisfaction problem. This allows for utilising modern constraint and SMT solvers. We optimised our refinement checker to efficiently analyse a large number of mutated models. Experiments showed that our optimisations reduced the runtimes by up to 90% compared to our first implementation.

Despite its efficiency, our notion of refinement is not completely satisfying the needs of model-based mutation testing. A more suitable conformance relation is ioco. However, previous work has shown that ioco checking is rather demanding in terms of runtime and memory consumption. Therefore, we use our optimised refinement check as a preprocessing step for the efficient computation of an under-approximation of an ioco test suite. Instead of performing a full ioco check between the original and a mutated model, we first perform a refinement check. Only in case of non-refinement, the ioco check is initiated from the point where non-refinement has been detected. In this way, the subsequent ioco check is more target-oriented.

Our test case generator has been integrated into a tool chain that generates test cases from UML state machines. It has been applied to two case studies conducted with industrial partners from the automotive domain in the context of two research projects.

Keywords: Test Case Generation, Model-Based Testing, Mutation Testing, Conformance, Refinement, Input-Output Conformance (ioco), Action Systems, Constraint Solving, SMT Solving.

Kurzfassung

Software ist allgegenwärtig in unserem täglichen Leben. Computer und eingebettete Systeme finden sich in Telefonen, Autos, Eisenbahn-Signalanlagen, etc. In manchen dieser Anwendungen ist ein Funktionsfehler unangenehm, in anderen lebensbedrohlich. Eine gründliche Verifikation und Validierung von Softwaresystemen ist deshalb unumgänglich. In der Praxis geschieht dies meist durch Softwaretesten.

In den letzten Jahren hat sich modellbasiertes Testen als vielversprechender Ansatz zum Softwaretesten erwiesen, weil es sich für automatisierte Testfallgenerierung eignet. Der Test-Ingenieur erstellt ein formales Modell des zu testenden Systems und verwendet Tools um automatisch Testfälle abzuleiten, anstatt diese manuell zu entwerfen. Ein wichtiger Punkt im modellbasierten Testen ist die Wahl des Testkriteriums, das angibt welche Testfälle generiert werden sollen. In dieser Arbeit folgen wir einem fehlerorientierten Ansatz. Das originale Testmodell wird syntaktisch abgeändert um so eine Menge von mutierten Modellen zu erzeugen. Wir überprüfen die Konformität zwischen dem originalen und den mutierten Modellen. Im Falle von Nichtübereinstimmung generieren wir automatisch einen Testfall der das unterschiedliche Verhalten des Mutanten aufzeigt. Wenn diese Testfälle auf dem zu testenden System ausgeführt werden, erkennen sie ob eines der fehlerhaften Modelle anstelle des korrekten, ursprünglichen Modells implementiert wurde. Somit decken die generierten Testfälle alle Fehler die durch die Mutationsoperatoren beschrieben werden ab und haben auch gute Chancen ähnliche Fehler zu finden.

Als Formalismus zur Modellierung verwenden wir *Action-Systeme*. Diese eignen sich gut um reaktive und nichtdeterministische Systeme zu modellieren. Ein Hauptfaktor für die Qualität und Effizienz von modellbasiertem Mutationstesten ist die zu Grunde liegende Konformitätsprüfung, die auch den Hauptfokus dieser Arbeit darstellt. Wir haben einen effizienten Testfallgenerator für Action-Systeme entworfen und implementiert. Er basiert auf zwei Konformitätsrelationen: *Refinement* und *Input-Output Conformance* (ioco).

Zur Überprüfung von Refinement haben wir eine prädikative Semantik von Action-Systemen definiert, die einem Constraint-Satisfaction-Problem ähnelt. Dies erlaubt es uns moderne Constraint- und SMT-Solver zu verwenden. Wir haben unseren Refinement-Checker optimiert um effizient eine große Menge von mutierten Modellen verarbeiten zu können. Experimente haben gezeigt, dass unsere Optimierungen die Laufzeiten im Vergleich zur Basisimplementierung um bis zu 90% reduzieren konnten.

Trotz der hohen Effizienz kann unsere Refinement-Relation die Anforderungen für modellbasiertes Mutationstesten nicht zur Gänze erfüllen. Eine passendere Konformitätsrelation ist ioco. Vorherige Arbeiten haben allerdings gezeigt, dass die Überprüfung auf ioco-Konformität höhere Laufzeiten und größeren Speicherbedarf mit sich bringt. Deshalb verwenden wir unseren optimierten Refinement-Check als Vorverarbeitungsschritt für die effiziente Berechnung einer Unterapproximation der ioco-Testsuite. Anstelle eines vollständigen ioco-Checks zwischen dem originalen und einem mutierten Modell, führen wir zuerst einen Refinement-Check durch. Nur im Falle einer Verletzung der Refinement-Relation wird ein ioco-Check angestoßen. Dieser beginnt an dem Punkt wo der Refinement-Verstoß identifiziert wurde. Auf diese Weise ist der nachfolgende ioco-Check zielgerichteter.

Unser Testfallgenerator wurde in eine Toolchain integriert, die Testfälle von UML-Zustandsdiagrammen generiert und wurde in zwei Fallstudien angewendet. Diese stammen aus der Automobilbranche. Sie wurden in Zusammenarbeit mit zwei Industriepartnern im Rahmen von zwei Forschungsprojekten durchgeführt.

Schlagworte: Testfallgenerierung, Modellbasiertes Testen, Mutationstesten, Konformität, Refinement, Input-Output Conformance (ioco), Action-Systeme, Constraint-Solving, SMT-Solving.

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)

Acknowledgements

I would like to thank numerous people who supported me while I was writing this thesis and accompanied me during my doctoral studies.

First of all, I express my gratitude to my supervisor Bernhard Aichernig. Long before this dissertation, he sparked my interest in software testing. He is a good teacher, who always has an open ear and motivating recommendations.

Furthermore, I am grateful to Robert Hierons for reviewing this thesis.

I thank all co-authors of the papers that formed the basis of this work, my colleagues of the research projects MOGENTES, TRUFAL, and MBAT, as well as my colleagues at the institute. Furthermore, I thank Matthias Kegele and Martin Tappler for their assistance.

I would like to address special thanks to my partner Robert for his support and understanding – especially in the final writing phase. Last but not least, I want to thank my parents Rupert and Helene and my whole family for keeping me grounded.

This work has been financially supported by the Austrian Research Promotion Agency (FFG) via the project TRUFAL (project number 829583), by the EU FP7 project MOGENTES (ICT-216679), and by the ARTEMIS Joint Undertaking MBAT (grant agreement number 269335).

Elisabeth Jöbstl
Graz, Austria, April 2014

Danksagung

Ich möchte mich herzlich bei allen bedanken, die mich während des Verfassens dieser Arbeit sowie im gesamten Doktoratsstudium unterstützt haben.

Insbesondere gilt mein Dank meinem Betreuer Bernhard Aichernig. Lange bevor ich meine Dissertation begann, hat er mein Interesse für das Softwaretesten geweckt. Er ist ein guter Lehrer, der immer ein offenes Ohr hat und motivierende Worte findet.

Des weiteren danke ich Robert Hierons dafür, dass er diese Arbeit begutachtet hat.

Ich bedanke mich bei allen Co-Autoren der Publikationen, die im Zuge dieser Arbeit entstanden sind. Des weiteren danke ich meinen Kollegen aus den Forschungsprojekten in denen ich mitwirken durfte. Ein großer Dank gilt auch all meinen Kollegen am Institut für Softwaretechnologie und den Studenten, die ich betreuen durfte und die dadurch einen Teil zu dieser Arbeit beigetragen haben.

Ganz besonderer Dank gilt meinem Partner Robert für sein Verständnis und für seine Unterstützung besonders während des Verfassens dieses Dokuments. Nicht zuletzt möchte ich meinen Eltern Rupert und Helene sowie meiner gesamten Familie für ihren Rückhalt und die moralische Unterstützung danken.

Diese Arbeit wurde finanziell unterstützt von der Österreichischen Forschungsförderungsgesellschaft (FFG) im Rahmen des Projekts TRUFAL (Projektnummer 829583), vom EU-FP7-Projekt MOGENTES (ICT-216679) und durch das ARTEMIS-Projekt MBAT (Kennziffer 269335).

Elisabeth Jöbstl
Graz, Österreich, April 2014

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Abbreviations

AGSL	Activity and Guard Specification Language
API	Application Programming Interface
CAS	Car Alarm System
CLP	Constraint Logic Programming
clpfd	Constraint Logic Programming over Finite Domains
CSP	Communicating Sequential Processes
EFSM	Extended Finite State Machine
FDR	Failures-Divergence Refinement
FSM	Finite State Machine
IDE	Integrated Development Environment
ioco	Input-Output Conformance
IOTS	Input Output Transition System
LTS	Labelled Transition System
MBT	Model-Based Testing
OCL	Object Constraint Language
OOAS	Object-Oriented Action System
RAISE	Rigorous Approach to Industrial Software Engineering
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
STS	Symbolic Transition System
SUT	System Under Test
TLA	Temporal Logic of Actions
UML	Unified Modelling Language
UTP	Unifying Theories of Programming
VDM	Vienna Development Method

1 Introduction

1.1 Motivation

Quality assurance is an important task in every engineering discipline – also in software development. Today, software is omnipresent in our everyday life. Computers and embedded systems are used naturally in telephones, entertainment electronics, cars, railway signalling systems, air traffic control systems, etc. Depending on the application domain, a malfunctioning can be inconvenient or even life-threatening for the users. For the manufacturers, incorrect system behaviours potentially entail enormous costs.

For example, the Ariane 5 rocket got uncontrollable and had to be exploded just 40 seconds after initiation of the flight sequence in 1996. The software system calibrated for the Ariane 4 rocket had been reused without proper testing. This fault caused costs in the amount of \$ 370 million [83]. Another tragic example for serious consequences of a software failure comes from the medical domain. Until 2005, a therapy planning software at the National Cancer Institute of Panama City caused at least 18 deaths. Although incorrect data sequences have been input to the software, it did not alert the user and calculated improper dosages of radiation for cancer patients [47]. These are just two examples out of a long list of exemplary software failures. Many rankings of the worst software bugs can be found on the internet [97, 145] – pointing out the need for high quality standards in software engineering.

Quality assurance in software engineering is a challenging and labour-intensive task. In practice, it is mostly conducted via software testing. Already in the 1970s, it was known that about 50% of the elapsed time and more than 50% of the total costs of a software project are spent on testing. Decades later, this still holds true [162]. Furthermore, the later a software failure is detected, the higher the costs for fixing it [108]. Testing is not only expensive, the costs for testing are also often underestimated and poorly planned [55]. Hence, tools and techniques to assist testers are demanded by industry. Software testing consists of three main tasks: test design, test execution, and test evaluation. In general, each task can be supported or even fully automated by testing tools [23]. In this work, we develop a formal approach to software testing and evaluate its applicability in an industrial setting. We follow a model-based approach that we combine with mutation testing to automatically generate a high-quality test suite.

1.2 Model-Based Mutation Testing

Model-based testing has become a well-known technique to automate test case generation [194, 195, 80, 113, 56]. Instead of designing many individual test cases, the test engineer creates a formal model of the System Under Test (SUT). It describes the expected behaviour of the SUT and is used as input to model-based testing tools that automatically derive test cases. A crucial matter in model-based testing is the choice of the test criterion. It specifies which test cases shall be generated and hence, has a great influence on the quality of the resulting test suite. Exhaustive testing, i.e., using all of the test cases that can possibly be created from the test model, is impractical for real-world applications. Examples for commonly used test criteria are coverage criteria (e.g., used in [201]), random traversals (e.g., [193, 158]), equivalence classes (e.g., [153]), or specified testing scenarios (so-called test purposes, e.g., [126, 198]). As illustrated in Figure 1.1, we follow a fault-centred approach, i.e., use mutations for test case generation. The original test model is syntactically altered to produce mutated models that represent a set of faults. We then automatically generate test cases that *kill* the model mutants, i.e., reveal their non-conforming behaviour. This is accomplished by a conformance check between the original and the mutated models. A particular feature of the generated test suites is their fault coverage. The generated tests will detect whether one of the given faulty models has been implemented instead of the correct,

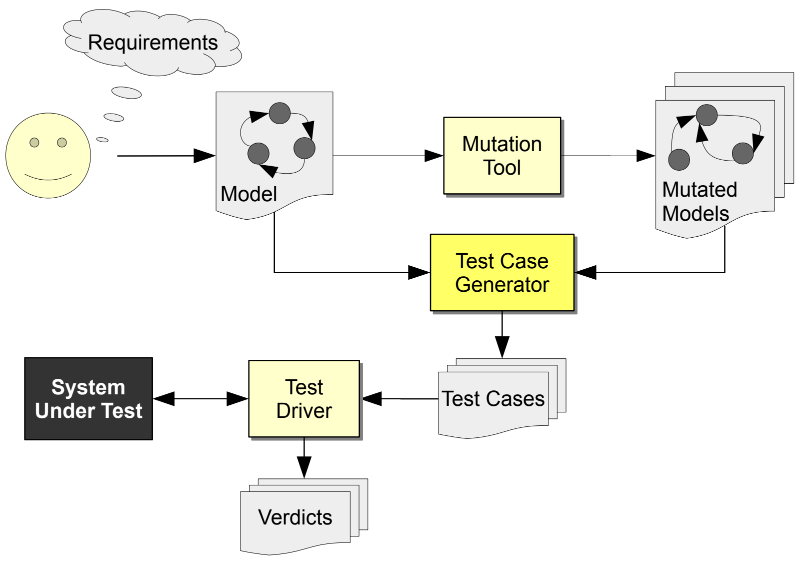


Figure 1.1: Overview of model-based mutation testing.

original model. Hence, the generated test suite covers all of the failures caused by the model mutation operators and has a high chance of covering many additional similar failures.

1.3 Problem Statement

The quality and performance of model-based mutation testing is influenced by two main factors. One issue concerns the required conformance check. Usually, industrial-sized systems show complex behaviour comprising a large-scale or even infinite state space. This poses a challenge for the conformance check, which has to compare the behaviour of the system with its mutants. We identified that the choice of an appropriate conformance relation is essential. First of all, this choice depends on the kind of systems to be tested and it affects the quality of the generated test suite. Finally, it has a great influence on the performance of the conformance check and hence on the overall test case generation. The importance of an efficient conformance check is reinforced by the nature of mutation testing. It typically involves a high number of mutations, i.e., the conformance check has to be performed numerous times for the generation of one test suite.

This directly leads us to the second main influencing factor of model-based mutation testing: the choice of the fault models, i.e., model mutations. On the one hand, they need to compose a representative set of faults in order to induce a high-quality test suite. On the other hand, they should not contain redundancies that make model-based mutation testing inefficient. The set of modelled faults is influenced by the modelling formalism and modelling style: depending on the modelling language and more specifically on the elements actually used in the model, different mutation operators are applicable leading to varying model mutations.

Each individual problem forms an own area of research. In this thesis, we focus on the conformance check. The modelling formalism (action systems [28]) and mutation operators were predetermined.

1.4 Thesis Statement

The applicability of model-based mutation testing can be improved by optimising the performance of the underlying conformance check. This can be established by choosing an appropriate conformance

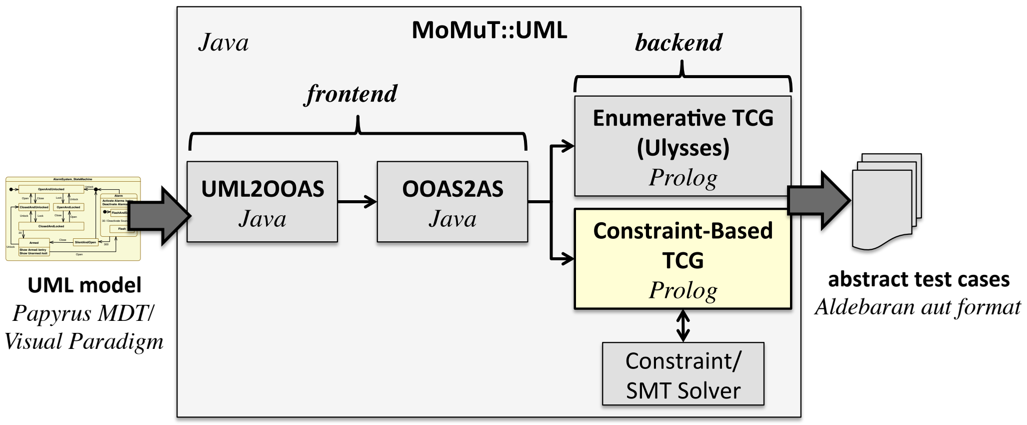


Figure 1.2: Architecture of the MoMuT::UML tool chain.

relation, and by the application of symbolic techniques utilising modern constraint/SMT solvers for the implementation of the conformance check.

1.5 Research Context

This work has been conducted within several research projects. Early work and some foundations were established throughout the European project MOGENTES (Model-based Generation of Tests for Dependable Embedded Systems)[1]. Ten partners from academia and from industry, which were located in six European countries, participated in the project. MOGENTES aimed for significantly enhancing verification and testing of embedded systems by automated test case generation. One of the approaches followed was model-based mutation testing.

MOGENTES initiated the Austrian follow-up project TRUFAL (TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation)[2]. TRUFAL is funded by FFG (Österreichische Forschungsförderungsgesellschaft)[3], and settled in the programme FIT-IT (Forschung, Innovation und Technologie für Informationstechnologien). Four partners contribute to the project: two research partners (AIT Austrian Institute of Technology and Graz University of Technology) and two industrial partners (AVL List GmbH and THALES Austria GmbH). The main parts of this thesis have been conducted within TRUFAL.

The tools and concepts developed in the course of the TRUFAL project are to be used and integrated in two further projects: MBAT (Combined Model-based Analysis and Testing of Embedded Systems)[4] and CRYSTAL (CRitical sYSTem engineering AcceLeration)[5]. MBAT has also financially supported this work.

1.5.1 The MoMuT::UML Tool Chain

Within the MOGENTES and TRUFAL projects, the MoMuT::UML tool chain for automated model-based mutation testing from Unified Modelling Language (UML) models has been developed. Figure 1.2

[1]`http://www.mogentes.eu` (last visit 2014-04-18)
[2]`http://trufal.wordpress.com` (last visit 2014-04-18)
[3]`https://www.ffg.at/` (last visit 2014-04-18)
[4]`https://www.mbat-artemis.eu` (last visit 2014-04-18)
[5]`http://www.crystal-artemis.eu` (last visit 2014-04-18)

gives an overview of MoMuT::UML's architecture, which consists of several components that form a *frontend* and a *backend*.

The frontend deals with model transformations to bring the input model into a format suitable for the backend, i.e., the actual test case generator. First, the UML model is transformed into a labelled and Object-Oriented Action System (OOAS) [140]. This executable intermediate representation has a formal semantics and is based on Object-Oriented Action System (OOAS) as introduced in [45]. This transformation is performed by the *UML2OOAS* component, which also implements the mutation operators for UML state machines. Subsequently, the resulting OOAS representing the original UML model and the set of OOAS models produced by applying the mutation operators are translated into non-object-oriented, but still labelled action systems, which are an extension of Back's action system formalism [28]. This second transformation is implemented in the *OOAS2AS* component. Both transformations were developed in the MOGENTES project and revised for further usage in the TRUFAL project. Note that both of these transformations are preliminaries for this work. The UML2OOAS component was developed by AIT Austrian Institute of Technology Vienna. The OOAS2AS component was developed by Willibald Krenn at Graz University of Technology. For details on the transformations in the frontend, we refer to Krenn et al. [140].

The action system models generated by the frontend, serve as input for the test case generation backend. In the MOGENTES project, an enumerative, mutation-based test case generator called *Ulysses* has been developed. It is a conformance checker for action systems and performs an explicit forward search of the state spaces. Thereby, the action systems generated by the frontend are explored. This process yields the Labelled Transition System (LTS) of the UML model. The conformance relation used is Input-Output Conformance (ioco) [191]. Ulysses has been developed by Harald Brandl at Graz University of Technology. For further information on the underlying theory and techniques, we refer to the following publications [10, 7, 9, 8, 50].

Ulysses was developed with a special focus on hybrid systems, i.e., systems consisting of discrete and a continuous behaviour. The discrete part modelling the controller of such systems was assumed to be rather simple. For complex discrete models, like the ones generated from UML state machines, experiments have shown that the performance of the explicit conformance checker Ulysses is lacking [10]. Especially, the tool suffers from a high memory consumption. Therefore, an alternative backend exploiting constraint/SMT solving has been developed in the course of this thesis. This component is highlighted in yellow in Figure 1.2. By setting the action system language and its semantics, MoMuT::UML fixes prerequisites for the test case generator developed in this work.

1.6 Use Cases

Throughout this thesis, we use two industrial use cases from the above described projects.

1.6.1 Car Alarm System

One use case in the MOGENTES project was from the automotive domain: a Car Alarm System (CAS). The following requirements describe the system:

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 unauthorised person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

R3 *Deactivation.* The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

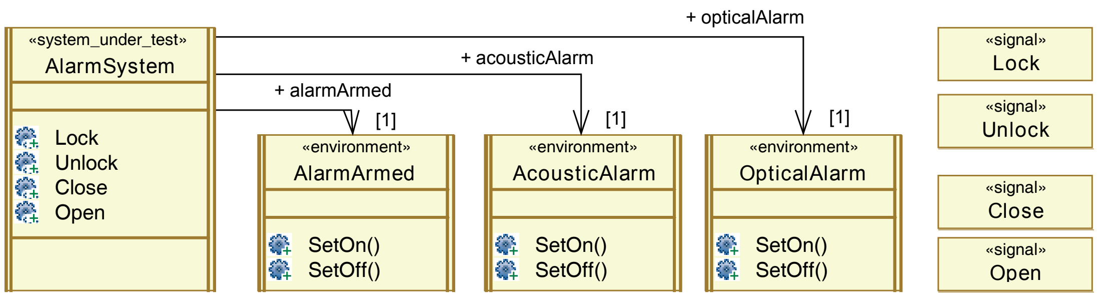


Figure 1.3: The testing interface of the CAS.

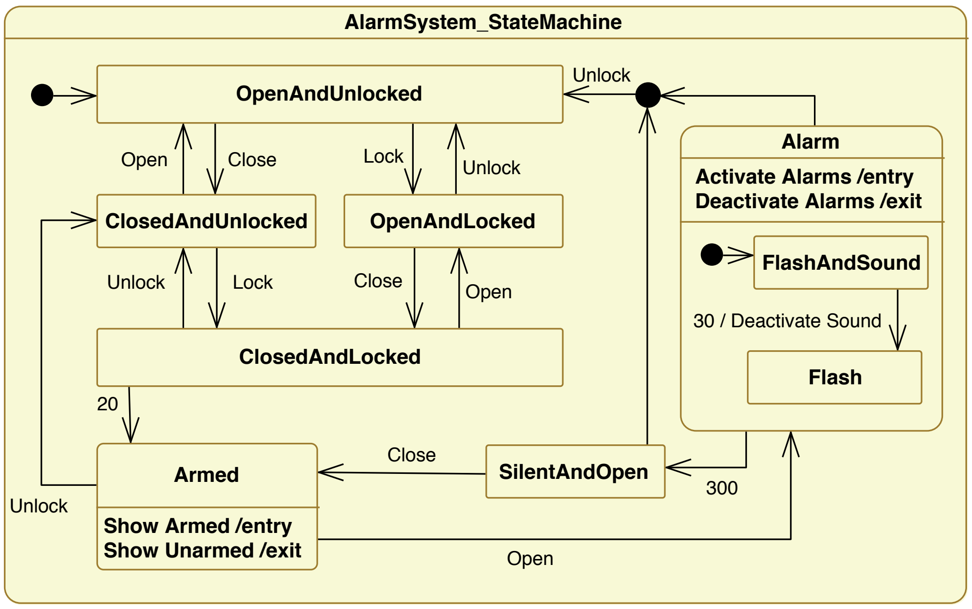


Figure 1.4: UML state machine of the CAS.

In the following, we illustrate the behaviour of the CAS by means of a UML model, which was provided by our project partner AIT. It comprises four classes and four signals as can be seen in Figure 1.3. The class *AlarmSystem* is tagged as *System Under Test (SUT)*. It receives the specified signals *Lock*, *Unlock*, *Close*, and *Open*. These signals are possible inputs to the SUT and are *controllable* by the tester. Furthermore, the *AlarmSystem* may call methods of the three classes *AlarmArmed*, *AcousticAlarm*, and *OpticalAlarm* representing the system's *environment*. These methods represent outputs from the SUT. They are *observable* by the tester.

Figure 1.4 illustrates the behaviour of the CAS by means of a UML state machine. Starting at state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car. As specified in requirement R1, the system is armed after 20 seconds in *ClosedAndLocked*. Upon entry of the *Armed* state, the method *AlarmArmed.SetOn* is called. Upon leaving the state (either by unlocking the car or by opening a door), *AlarmArmed.SetOff* is called. Similarly, 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 alarms are turned off. Note that the order of these two events is not specified, neither for enabling nor for disabling the alarms. Hence, the system is not deterministic. When leaving the alarm state after a timeout (cf. requirement R2) the system returns to the *Armed* state only if it receives a close signal. Turning off the acoustic alarm after 30 seconds, as specified in requirement R2, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state. As the CAS is simple, but still not trivial, it is used for demonstration throughout this thesis.

1.6.2 Particle Counter

A second use case from the automotive domain was provided for the TRUFAL project by the industrial partner AVL. It is a particle counter device[6], which measures the particle number concentration in the exhaust gases of combustion engines. More precisely, the control logic of this device shall be tested.

The user can choose between continuously measuring the current concentration and accumulating the total particle counts. During the measurement, the ratio by which the exhaust gas is mixed with particle-free dilution air can be adjusted by a factor between 1 and 7. Additionally, there is a command to measure pure, particle-free air to check whether the sensors are correctly calibrated. Other commands are provided for necessary maintenance tasks like a leakage test, a response check, or for purging the sampling line.

In total, the particle counter distinguishes between eight different operating states that can be triggered by the used testing interface. They include two idle states (*Pause* and *Standby*) as well as states like *Measurement* for *Purging*. Additionally, there are two different communication modes: *Manual* for controlling the particle counter directly via the buttons at the device and *Remote* for controlling the system remotely via a client, which shall be tested. Furthermore, the system may switch from a *Ready* to a *Busy* status when processing a command.

The device receives commands from the user interface and shows its current state and each change between different internal modes. Commands from the user may be rejected due to several reasons: (a) the command may not be available in the current operating state, (b) the system may be in the wrong communication mode, or (c) the system may be busy. In each case, the system returns an appropriate error message.

Initially, the system is idle (in operating state *Pause*), *Ready* to accept commands, and expects *Manual* communication. In order to receive commands via the testing interface, it has to be set into the *Remote* communication state.

For example, a possible scenario would be to start the measurement, adjust the dilution ratio, switch the measurement method to accumulating the total particle counts, and turn the measurement off again. Including the operations for switching the system into the right modes and observing all the output events from the system, a test case performing this scenario consists of 17 steps, i.e., input and output events.

1.7 Contributions and Publications

1.7.1 Contributions

The main contributions of this thesis can be summarised as follows.

- We designed and implemented an efficient conformance checker for action system models. The used conformance relation is refinement, which allows for non-determinism. For efficiency reasons, our refinement check is based on constraint/SMT solving techniques.
- To encode action systems into constraints that can be processed by constraint/SMT solvers, we defined a formal predicative semantics for action systems. We proved that this semantics is equivalent to the standard semantics, which is defined via weakest pre-conditions.

[6]`https://www.avl.com/particle-counter` (last visit 2014-04-18)

- We furthermore enhanced the efficiency of our refinement checker by several optimisation techniques. The evaluation on two industrial use cases showed that the computation time of the refinement check could be reduced by several orders of magnitude.
- We extended our refinement checker to an actual test case generator, where the counterexamples from the conformance check serve as basis for the test cases.
- We integrated our refinement-based test case generator into an existing tool chain, which allows for model-based mutation testing using UML models. Therefore, we had to enhance our test case generator by more complex modelling language constructs.
- Additionally to our refinement checker, we added an Input-Output Conformance (ioco) checker to our tool set. The motivation was to strengthen the generated test suites as refinement does not perfectly fit the needs of model-based testing. However, ioco checking is more expensive than refinement checking. In order to cope with complex, industrial-sized models, we approximate an ioco check by combining our refinement check with a subsequent ioco check.
- To avoid redundancies in the generated test suites, we introduce a check whether existing test cases already kill a mutated model. Furthermore, we combine our mutation-based test case generation approach with random test cases.
- All of our intermediate implementations as well as the final tool were applied on two industrial use cases. The test suites generated by the final tool were evaluated by assessing their fault detection capabilities on a set of faulty implementations.

1.7.2 List of Publications

Partly, the work presented in this thesis has already been published in international workshops, conferences, and journals. All of these publications have been formally peer reviewed.

Main Publications

The following publications directly form the basis of this thesis:

MBT 2010 [14] This is the first paper dealing with the work conducted within this thesis. It presents our basic approach for refinement checking of action systems. The paper was mainly written by myself with contributions from Bernhard Aichernig on the action systems and the refinement relation. He also proofread and revised the paper. I presented this work at the 7th Workshop on Model-Based Testing in Tallinn, Estonia at March 25th 2012.

CSTVA 2012 [15] This paper deals with problems I experienced during the implementation of our refinement checking approach for action systems. Bernhard Aichernig contributed text on action systems and refinement. Again, he revised the paper. I presented this work at the 4th Workshop on Constraints in Software Testing, Verification, and Analysis in Montreal, Canada at April 21st 2012.

QSIC 2012 [13] This publication contains three optimisation techniques for our previously presented refinement checker for action systems. Again, I did the implementation and experimentation as well as the main paper writing. Bernhard Aichernig was concerned with the introduction and conclusion. I presented this work at the 12th International Conference on Quality Software, which was very competitive with the low acceptance rate of only 17.6%. The conference was held in Xi'an, China in August 2012.

TAP 2013 [16] This publication is based on joint work with Bernhard Aichernig and Matthias Kegele. It introduces two further improvements on efficiency in our refinement check. Moreover, the paper relates our refinement checker implemented in Prolog with a re-implementation in Scala, which was written by Matthias Kegele. The experiments with the Prolog implementation were conducted by me. Matthias Kegele provided results for his Scala implementation. The paper was mainly written by me with contributions from Bernhard Aichernig. I presented our work at the 7th International Conference on Tests and Proofs in Budapest, Hungary at June 18th 2013.

SCP journal article [17] We were invited to extend our QSIC 2012 conference paper [13] for a special issue of the *Science of Computer Programming* (SCP) journal. This article gives a comprehensive overview of our refinement checking approach and all implemented optimisations. We conducted a second, industrial case study used for the evaluation of our optimisations. Furthermore, we presented how we construct test cases from the counterexamples witnessing non-refinement. This article is mainly based on my own work. However, Stefan Tiran developed the action system model for the additional case study and helped with the experiments. Furthermore, Bernhard Aichernig revised the paper.

Additionally to the above publications, a research abstract [132] has been prepared at the beginning of this thesis. However, it was not formally published. I presented it at the Doctoral Symposium of FM 2011 in Limerick, Ireland on June 20th 2011.

Other Related Publications

Our following publications are also related to the topic of this thesis:

ICST 2010 [133] This paper is a very condensed version of my master's thesis [131], which dealt with model-based testing via symbolic execution of input-output transition systems. I presented the paper at the 3rd International Conference on Software Testing, Verification and Validation in Paris, France at April 6th 2010.

FMCO 2009 [7] This paper dealt with Harald Brandl's work on the enumerative ioco checker and test case generator Ulysses. I implemented and described parts of the test case extraction approaches and helped with the experiments.

UML & FM 2010 [9] This paper also dealt with Harald Brandl's work on the Ulysses tool. I contributed to the experiments, related research, and the presentation of the test case generation tool.

ICST 2011 [8] This work described test case generation with the Ulysses tool. My contributions comprise parts of the implementation of the test case generation approaches and assistance with the experiments. I presented the paper at the 4th International Conference on Software Testing, Verification and Validation in Berlin, Germany at March 21st 2011.

SAFECOMP 2011 [176] The main authors are Rupert Schlick and Wolfgang Herzner from AIT Vienna. I contributed text on related work and the description of the test case generation methodology.

STVR 2014 [10] This article in the journal of Software Testing, Verification and Reliability resulted from an invitation to extend our publication from ICST 2011 [8]. I was involved in the design of the newly added experiments, which were then mainly performed by Stefan Tiran and to a smaller extent by me. I also contributed to the evaluation and description of the experimental results as well as to more detailed descriptions of the test case selection strategies including examples.

1.8 Structure of this Thesis

The rest of this thesis is structured as follows. Chapter 2 gives an overview of software testing and puts the work conducted in this thesis into context. Chapter 3 discusses conformance relations. It focuses on refinement and Input-Output Conformance (ioco), which are applied in this work. Chapter 4 presents model-based mutation testing. The modelling formalism used in this work are action systems, which are introduced in Chapter 5.

Chapter 6 presents our refinement checking approach for action systems and Chapter 7 reports on optimisations. Chapter 8 shows how test cases are constructed from the counterexamples of our refinement check and Chapter 9 reports on the extensions required to integrate our test case generation tool into the MoMuT::UML tool chain.

Chapter 10 extends our refinement check by an ioco check and compares our combined approach to pure ioco checking. Chapter 11 presents two optimisations of the generated test suites and reports on our final experimental results including test case execution and evaluation.

Chapter 12 gives a brief overview of related work and Chapter 13 summarises and concludes the thesis. Finally, it gives an outlook on future work.

2 Software Testing Background

In the following, we introduce the main terminology used in software testing required for this work.

2.1 Verification and Validation

Quality assurance in software engineering is divided into verification and validation [23]:

Definition 2.1 (Verification)
Verification checks whether the product is built in the right way, i.e., if its functions are implemented correctly according to a given specification.

Definition 2.2 (Validation)
Validation addresses the evaluation of the software according to the needs of the users. It asks, whether the right product has been built.

It can happen that a product is successfully verified, but that validation fails. Often, this can be led back to misunderstandings between the developers and the users during the collection of the requirements, which serve as basis for the specifications used in verification.

In this thesis, the focus lies on verification. There exist two major categories of software verification: static and dynamic verification techniques:

Definition 2.3 (Static Verification)
Static verification comprises verification techniques that do not require the actual execution of the program.

Examples for static verification techniques are model checking [71], static code analysis [163, 48, 66], or correctness proofs, e.g., in the style of Hoare logic [117].

Definition 2.4 (Dynamic Verification)
Dynamic verification comprises verification techniques that require program execution.

2.2 Software Testing

Dynamic verification basically refers to software testing, which is the topic of this thesis. In order to give a definition of testing, the following three terms need to be defined:

Definition 2.5 (Fault)
A software fault is a *“static defect in the software”* [23].

Definition 2.6 (Error)
A software error is *“an incorrect internal state that is the manifestation of some fault”* [23].

Definition 2.7 (Failure)
A software failure is an *“external, incorrect behaviour with respect to the requirements or other description of the expected behaviour”* [23].

The relation between these three terms is depicted in Figure 2.1. A fault is some wrong expression in the source code. In most cases, it is caused by human mistakes, e.g., by a misunderstanding of the requirements. This fault is not necessarily a problem. If it is never *reached*, i.e., executed, it will not become an issue. In case it is executed, it must cause the program to enter an incorrect state in order to produce an error. This is also referred to as *infection*. A failure only occurs, if the wrong program state *propagates* to an incorrect, observable output. Hence, only failures can be observed during testing.

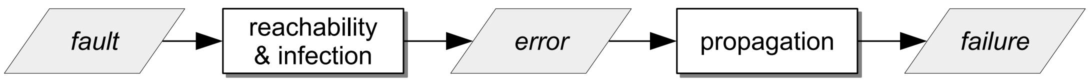


Figure 2.1: The relation between faults, errors, and failures.

Definition 2.8 (Software Testing)
Software testing is the process of executing a program with the intent of revealing failures.

This definition is based on Glenford Myers [162] and points out the nature of testing. The objective of testing is not showing correctness, but revealing as many failures as possible. This directly reflects Dijkstra's opinion on testing: *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"* [81]. Exhaustive testing, i.e., checking all possible inputs and the expected outputs of a program, is not possible in general. Hence, revealing all failures of a program is infeasible. Testing can only increase the confidence into a system. It cannot prove correctness.

It turned out that formal, static verification techniques and testing are complementary tasks [100]. Proofs always rely on assumptions, e.g., assumptions about the execution environment. Testing is the only way to verify a *running* system and to check the underlying assumptions. For example, the binary search algorithm of the Java JDK 1.5 library has been proven correct, but still contained a very subtle bug that was detected nine years later [43]. In this case, integer overflows have not been considered. Hence, the wrong assumption was that integers are unbounded, which holds in mathematical theory. However, hardly any programming language provides unbounded integer data types.

Furthermore, in most cases, full formal verification of a piece of software is not feasible. Possible reasons are the increasing complexity of software systems, the lack of highly-educated staff, or monetary restrictions. In such cases, testing is a viable alternative to full verification if it is systematic and automated. In 1995, Gaudel argued that *"testing can be formal, too"*. She presented a formal framework for testing that also included hypotheses on the system for the systematic selection of test cases [99].

However, not only full verification, but also testing is a challenging and labour-intensive task: about 50% of the elapsed time and more than 50% of the total costs of a software project are spent on testing [162]. Thus, as many software testing activities as possible should be automated. Software testing consists of three main tasks: test design, test execution, and test evaluation. In general, each task can be supported or even fully automated by testing tools [23]. In this work, we focus on the automation of the test design, i.e., the creation of a meaningful set of test cases. Generally, each test case consists of inputs and outputs, which are defined in the testing interface:

Definition 2.9 (Testing Interface)
The testing interface specifies the interaction between the SUT and its environment. It states how to send inputs to the SUT and declares all outputs that can be emitted by the SUT. In the context of testing, inputs are referred to as *controllable events* as inputs are controlled by the tester. Outputs are also denoted as *observations* (or *observable events*) as they are observable by the tester.

Depending on the type of the SUT, the shape of the test cases varies. The literature often distinguishes between transformational and reactive systems.

Definition 2.10 (Transformational System)
Transformational systems take an input, process it by performing computations, and return the resulting output [110].

Definition 2.11 (Reactive System)
Reactive systems are repeatedly prompted by the environment and respond continuously to external inputs. Hence, they are maintaining an ongoing interaction with their environment instead of calculating some final result [110].

Examples for reactive systems are operating systems, communication networks, or avionics systems. In most cases, reactive systems involve concurrency and non-determinism.

Definition 2.12 (Non-Deterministic System)
A non-deterministic system may produce different outputs for the same inputs.

Now, the shape of test cases can be defined for the different kinds of systems:

Definition 2.13 (Test Case)

- For transformational systems, a test case is an input/output pair.
- Test cases for reactive systems are sequences of controllable events (inputs) and expected observations (outputs).
- Test cases for non-deterministic, reactive systems are not only linear sequences of controllable and observable events, but may include branching over several possible observations.

Test cases are categorised into positive and negative test cases:

Definition 2.14 (Positive Test Case)
A positive test case tc^+ only contains expected behaviour as specified in the test model M, i.e., $tc^+ \subseteq M$. An SUT must not show any observation that is not specified in tc^+.

Definition 2.15 (Negative Test Case)
A negative test case tc^- explicitly states what is not expected. Hence, it is not included in the test model M, i.e., $tc^- \nsubseteq M$. An SUT must not show any observations specified as undesired in tc^-.

As the majority of other testing approaches, we focus on positive test cases. In the rest of this work, we always mean positive test case, although we may simply write test case. Independently of this distinction, (un)expected outputs are always an essential part of a test case. However, it is not always easy to determine the expected outputs of a system. This is also referred to as the *oracle problem.*

Definition 2.16 (Test Oracle)
An oracle provides the correct, expected system outputs.

In manual test design, the test engineer is the oracle and decides what is expected from the SUT. Further examples for test oracles are mathematical formulas or formal models that specify the expected outputs.

The outcome of the test design phase is a test suite:

Definition 2.17 (Test Suite)
A test suite denotes a set of test cases.

During test execution, the previously designed test cases are executed on the SUT:

Definition 2.18 (Test Run)
The execution of a test case on an SUT is referred to as test run, where the tester provides inputs specified in the test cases to the SUT and observes outputs from the SUT.

Finally, during test evaluation, it is decided whether a failure has occurred. The outcome of a test run is denoted as verdict:

Definition 2.19 (Verdict)
A verdict can be *fail*, *pass*, or sometimes also *inconclusive*. It is determined using the observations from test execution.

- **Fail:** Observations that do not correlate to the expected outputs specified in the test cases lead to a fail verdict and the SUT fails for the given test case. An SUT fails for a test suite if it fails for one of the test cases.

- **Pass:** If all observations emitted by the SUT comply with the expected outputs of the test case, the SUT passes the test case. An SUT passes a test suite if it passes all test cases.
- **Inconclusive:** Inconclusive verdicts state that the *test goal*, i.e., the functionality that should have been tested by the test case, could not be reached. This may happen due to non-determinism in the SUT. According to Definition 2.12, a non-deterministic SUT possibly delivers different outputs for the same input. If only one of the allowed outputs leads to the test goal, but the SUT chooses another one, the SUT behaves correctly, but the test goal cannot be reached any more.

Note that this definition of verdicts refers to positive test cases as defined in Definition 2.14.

The subsequent phase of fault localisation (finding the cause of a failure detected by testing), and repair (fixing the fault), are not part of software testing. They belong to *software debugging*. This work focuses on software testing. In particular, on the automation of test case generation for reactive systems. As reactive systems are often non-deterministic, we provide support for non-determinism. Of course, we can also deal with deterministic systems.

2.3 Software Testing Approaches

Many approaches to software testing exist. They can be classified by many characteristics. In the following, we present the ones most commonly used in the literature to set our approach into context. A very rough, but common classification is the distinction between white-box and black-box testing [162].

Definition 2.20 (Black-Box Testing)
Black-box testing techniques only know the interface of the SUT. Test cases can only be derived according to requirements documents, formal specifications, etc.

Definition 2.21 (White-Box Testing)
White-box testing techniques assume access to the internals of the SUT, i.e., the source code.

Later on, also *gray-box testing* approaches emerged, mixing concepts of black- and white-box testing [135]. Another way to classify software testing approaches is to distinguish between functional and non-functional testing [194].

Definition 2.22 (Functional Testing)
Functional testing checks whether the SUT correctly implements the functional requirements, e.g., a given protocol specification.

Definition 2.23 (Non-Functional Testing)
Non-functional testing aims at requirements such as robustness, timing, performance, or usability.

According to the stages in software development, it is furthermore distinguished between unit, module/-component, integration, system, and acceptance testing [194, 23].

Definition 2.24 (Levels of Software Testing)
- *Unit testing* is applied at the most detailed level. It tests small units of the system, e.g., individual functions or methods.
- *Module or component testing* is settled one level higher. It tests components of a system, e.g., a class or a group of logically connected classes.
- *Integration testing* focuses on the interoperability and consistency between groups of components.
- *System testing* is applied to the overall software product to be delivered.

- *Acceptance tests* are mostly conducted together with the end users to determine whether the software product fulfils the user requirements.

Note that all testing levels except for acceptance testing deal with verification. Acceptance testing aims for validation. Regarding test execution, we can distinguish between offline and online testing.

Definition 2.25 (Offline Testing)
In offline testing, test design and execution are separate tasks. A set of executable test cases is generated. Afterwards, this test suite is executed on the SUT.

An advantage of this methodology is that test cases can be reused. They are generated once and can be executed several times. This is beneficial for testing several implementations of the same system like in regression testing. Furthermore, test cases for offline testing can be generated before the system under test is implemented.

Definition 2.26 (Online Testing)
In online testing, test generation and execution are conducted on the fly, i.e., inputs are generated and directly delivered to the SUT. The outputs from the SUT are observed and compared with the expected outputs. Then, test input generation is continued.

Online testing is mainly applied to non-deterministic systems. It has the advantage that the tester does not need to be prepared for all possible outputs. It can continue test case generation for the actually produced output of the SUT.

The testing approach followed in this thesis can be classified as follows: we conduct *black-box testing* to verify the *functional* requirements on the *system level*. We automatically generate a test suite that is executed *offline*.

3 Conformance

Parts of this chapter are based on our STVR article [10].

In the context of verification (cf. Definition 2.1), testing is sometimes also referred to as *conformance testing*. In conformance testing, a SUT is tested to find possible violations of its specification. In 1991, the International Organization for Standardization (ISO)[7] published the ISO/IEC 9646 standard: *Information technology - Open Systems Interconnection - Conformance testing methodology and framework* [91]. In 1992, Tretmans [190] formalised the main concepts introduced in this standard.

The central question in conformance testing is:

When is a SUT (in)correct with respect to its specification?

The answer to this question is given by *conformance relations*. In order to formally relate a formal specification $s \in \mathcal{F}_s$ with an implementation $i \in \mathcal{I}$, it is assumed that each implementation i can be represented by a formal object $i_f \in \mathcal{F}_i$, such that the formalisms $\mathcal{F}_i$ and $\mathcal{F}_s$ are compatible. This is a common *test hypothesis* in the literature [38, 190]. Note that the concrete formal description of the implementation does not need to be known. The hypothesis only assumes its existence. In many cases, the formalisms for describing the expected behaviour and the implementation's behaviour are assumed to be the same, i.e., $\mathcal{F}_i = \mathcal{F}_s$. For example, the specification of a communication protocol can be expressed by a Labelled Transition System (LTS), which can also be used to describe the behaviour of protocol implementations. However, it is not necessary that both formalisms are the same, they just need to be compatible. Consequently, conformance relations can be defined as follows.

Definition 3.1 (Conformance Relation)
A conformance relation specifies in which cases an implementation is correct with respect to its specification. Formally, a conformance relation *conf* is a relation over formal descriptions of the implementation behaviour $\mathcal{F}_i$ and formal specifications $\mathcal{F}_s$: $\mathit{conf} \subseteq \mathcal{F}_i \times \mathcal{F}_s$.

In the literature, many conformance relations have been defined specifying different notions of correctness. Some conformance relations are *stricter* than others, i.e., they require stronger conditions to be fulfilled for conformance. One of the strictest conformance relations based on external observations is *observational equivalence*: given the same inputs, the same observable behaviour has to be emitted by both systems.

To mathematically define an equivalence relation, the following properties of relations are needed [75]:

Definition 3.2 (Reflexivity)
A binary relation $\mathcal{R}$ on a set P is reflexive iff $\forall a \in P : a\ \mathcal{R}\ a$.

Definition 3.3 (Symmetry)
A binary relation $\mathcal{R}$ on a set P is symmetric iff $\forall a, b \in P :\ a\ \mathcal{R}\ b \Rightarrow b\ \mathcal{R}\ a$.

Note that we use $\Rightarrow$ to denote logical implication.

Definition 3.4 (Transitivity)
A binary relation $\mathcal{R}$ on a set P is transitive iff $\forall a, b, c \in P :\ a\ \mathcal{R}\ b \wedge b\ \mathcal{R}\ c \Rightarrow a\ \mathcal{R}\ c$.

Definition 3.5 (Equivalence Relation)
An equivalence relation is a binary relation that is reflexive, symmetric, and transitive.

[7]`http://www.iso.org` (last visit 2014-04-18)

As already pointed out, equivalence is a very strict conformance relation. However, specifications should specify *what* shall be implemented and not *how* it shall be done. Hence, they should grant some implementation freedom. In other words, good specifications are more abstract than their concrete implementations. This requires conformance relations that are preorder or partial order relations [75]:

Definition 3.6 (Preorder Relation)
A preorder relation is a binary relation that is reflexive and transitive.

Hence, symmetry is given up in preorder relations. This enables a specification S to be more abstract than its implementation I. The implementation must fulfil the specification. However, the specification does not need to conform to the implementation. Hence, a preorder does not need to hold in both directions.

Partial order relations additionally require antisymmetry leading to a stricter ordering [75]:

Definition 3.7 (Antisymmetry)
A binary relation $\mathcal{R}$ on a set P is antisymmetric iff $\forall a, b \in P : \; a \, \mathcal{R} \, b \wedge b \, \mathcal{R} \, a \;\Rightarrow\; a = b$.

Definition 3.8 (Partial Order Relation)
A partial order relation is a preorder relation that is antisymmetric.

In the following, we discuss the two conformance relations that are relevant for this work: refinement and Input-Output Conformance (ioco). For a comprehensive overview of various kinds of conformance relations for different specification formalisms, we refer to Hierons et al. [113].

3.1 Refinement

Refinement relations are preorders or partial orders [64]. They were originally used in *program refinement*, which deals with the step-wise development of programs. The basic idea is to have initial programs that are successively replaced by improved, refined versions. Every refinement step is formally verified. This principle is used in the Vienna Development Method (VDM) [134], the Rigorous Approach to Industrial Software Engineering (RAISE) [107], or the B-method [1]. Instead of verifying possible refinements, *refinement calculi* are used to formally derive refined programs by following given refinement laws [34, 160]. For a comparison of these two concepts, we refer to the article of Litteck and Wallis [154].

Refinement can be divided into operational refinement and data refinement [73]:

Definition 3.9 (Operational Refinement)
Operational refinement deals with the refinement of behaviour, i.e., more abstract algorithms are replaced by more concrete ones with reduced non-determinism.

Definition 3.10 (Data Refinement)
Data refinement addresses different representations of the program's state, i.e., coarse data representations are refined by more detailed data structures.

Data refinement requires a mapping between abstract and concrete data. For details on data refinement, we refer to deRoever and Engelhardt [77]. Operational refinement and data refinement can be used in combination – like in Event-B [2] for example. In this work, we concentrate on operational refinement. Diverse definitions of operational refinement relations exist as there are various modelling formalisms and different semantic models at hand. The semantics of a programming or modelling language describes the meaning of the syntactical constructs available in that language.

There exist three main semantic models: *operational*, *denotational*, and *algebraic* semantics [119]. An operational semantics uses abstract mathematical machines, e.g., automata, to describe the operational behaviour of a language, i.e., the individual steps in its execution. For example, the operational semantics of Boolean expressions defines their evaluation order, e.g., from left to right. A denotational semantics maps each syntactical construct to some value in a mathematical domain. It describes only the effect of a computation, not how this computation is executed. For example, truth tables for Boolean expressions represent their interpretation function. It maps the operators and operands to their semantic truth values. Finally, algebraic semantics define the algebraic properties of a language by equational axioms. For example, Boolean algebra defines the algebraic semantics of Boolean expressions. According to these styles of semantics, different refinement relations have been defined. In the following, we give a brief overview of refinement relations. For more details, we refer to Cavalcanti et al. [64].

3.1.1 Axiomatic Refinement

Axiomatic refinement relates algebraic semantics. Informally, an implementation refines a specification if it satisfies all axioms of the specification. For example, all implementations of a stack must fulfil the axioms for a stack, e.g., $pop(push(Stack, Elem)) = Stack$ or $top(push(Stack, Elem)) = Elem$. This style of refinement is used in the RAISE method [107] for example.

The following refinement relations address denotational semantics.

3.1.2 Traces Refinement

The (denotational) semantics of reactive systems is often defined via *traces*. The traces of a system represent all possible sequences of events, i.e., synchronisations between the system and its environment. Refinement is defined as trace inclusion, i.e., all traces of the implementation I must be included in the set of traces of the specification S, i.e., $traces(I) \subseteq traces(S)$ [175]. Obviously, specifications must be complete. Consider a vending machine that may produce coffee or tea after insertion of a coin. A correct implementation I would show the traces $T_1 = \langle coin, coffee\rangle$ and $T_2 = \langle coin, tea\rangle$. An incomplete specification S_1 may only consider the tea functionality, i.e., only trace T_2. Traces refinement for the full implementation against the incomplete specification does not hold since $traces(I) = \{T_1, T_2\} \nsubseteq \{T_2\} = traces(S_1)$. However, S_1 conforms to I with respect to traces refinement, since $traces(S_1) = \{T_2\} \subseteq \{T_1, T_2\} = traces(I)$ holds.

3.1.3 Failures(-Divergences) Refinement

Additionally to the rather course traces semantics, failures and divergences have been introduced. Traces specify what a system *can* do, not what it *must* do. Therefore, refusal sets have been introduced to add information about which events a system may *refuse* to do. A *failure* relates a finite trace to its refusal set. *Failures refinement* extends traces refinement by additionally requiring failures inclusion, i.e., $traces(I) \subseteq traces(S) \wedge failures(I) \subseteq failures(S)$. Still, one behaviour cannot be specified yet: divergences or livelocks. *Divergences* are those traces, after which an infinite sequence of consecutive internal events can occur. *Failures-divergences refinement* is defined as $failures(I) \subseteq failures(S) \wedge divergences(I) \subseteq divergences(S)$. The traces inclusion is implied by this definition, i.e., the inclusion between traces do not need to be explicitly mentioned [175].

3.1.4 Weakest-Pre-condition Refinement

A common denotational semantics uses weakest pre-conditions, which were originally defined by Dijkstra [82]. In a weakest-pre-condition semantics, program statements are seen as predicate transformers, which map a given post-condition to the weakest pre-condition that the statement must fulfil in order to satisfy the post-condition. Refinement is defined via implication. An implementation I refines a specification S if and only if for all post-conditions Q, the weakest pre-condition of I is stronger than that of S, i.e., $\forall Q : wp(S,Q) \Rightarrow wp(I,Q)$ [63].

3.1.5 Relational Refinement

Another form of denotational semantics employs relations between pre- and post-states of statements. Intuitively, refinement is defined such that an implementation must not reach states that are not specified.

The state relations can be expressed as *pre-/post-condition* pairs and refinement has two requirements: pre-condition weakening and post-condition strengthening. The pre-condition of an implementation has to be weaker than the pre-condition of the specification and the post-condition of the implementation has to be stronger than that of the specification [63, 119]. The pre-/post-condition approach allows for incomplete specifications.

More generally, state relations can be represented by any kind of *predicates* and refinement is defined as implication over these predicates [119]: the implementation must imply the specification. In the rest of this work, the term *refinement* will refer to this kind of relational refinement:

Definition 3.11 (Refinement)
Let $\overline{v} = \langle x, y, \dots \rangle$ be the set of variables denoting observations before execution (the pre-state) and let $\overline{v}' = \langle x', y', \dots \rangle$ be the set of variables denoting the observations afterwards (post-state). The fact that an abstract specification S is refined by a concrete implementation I is denoted as $S \sqsubseteq I$. Given a predicative semantics for S and I, then

$$S \sqsubseteq I \;=_{df}\; \forall \overline{v}, \overline{v}' : I(\overline{v}, \overline{v}') \Rightarrow S(\overline{v}, \overline{v}')$$

Example 3.1. Again, consider a simple vending machine, which delivers tea or coffee. We express this via a variable bev representing the chosen beverage. It is assigned the value 1 for tea or the value 2 for coffee, i.e., $bev := 1 \,\Box\, bev := 2$, where the operator $\Box$ denotes non-deterministic choice. In terms of predicates, we have $S_1(\overline{v}, \overline{v}') = (bev' = 1 \vee bev' = 2)$, where $\overline{v} = \langle bev \rangle$ and $\overline{v}' = \langle bev' \rangle$. Analogously, we consider a system that always delivers tea, i.e., $bev := 1$ or in terms of predicates $S_2(\overline{v}, \overline{v}') = (bev' = 1)$ with $\overline{v} = \langle bev \rangle$ and $\overline{v}' = \langle bev' \rangle$. S_1 is refined by S_2, i.e., $S_1 \sqsubseteq S_2$ as we show in the following:

$$\begin{aligned}
& S_1 \sqsubseteq S_2 && \{\text{Definition 3.11}\} \\
&= \forall \overline{v}, \overline{v}' : (S_2(\overline{v}, \overline{v}') \Rightarrow S_1(\overline{v}, \overline{v}')) && \{\overline{v} = \langle bev \rangle\} \text{ and } \{\overline{v}' = \langle bev' \rangle\} \\
& && \{S_1(\overline{v}, \overline{v}') = (bev' = 1 \vee bev' = 2)\} \text{ and } \{S_2(\overline{v}, \overline{v}') = (bev' = 1)\} \\
&= \forall bev, bev' : ((bev' = 1) \Rightarrow (bev' = 1 \vee bev' = 2)) && \{\text{first-order predicate calculus}\} \\
&= \forall bev, bev' : (\neg(bev' = 1) \vee (bev' = 1 \vee bev' = 2)) && \{\text{first-order predicate calculus}\} \\
&= \forall bev, bev' : (true \vee bev' = 2)) && \{\text{first-order predicate calculus}\} \\
&= \forall bev, bev' : true && \{\text{first-order predicate calculus}\} \\
&= true
\end{aligned}$$

However, the other direction does not hold: S_2 is not refined by S_1, i.e., $\neg(S_2 \sqsubseteq S_1)$. The proof works analogously to above. $\Box$

In the above example, the observations were pre- and post-states, i.e., variable valuations before and after execution. However, also traces can be observations. In the following, we reconsider the vending machine example with traces instead of states.

Example 3.2. As already shown in Section 3.1.2, the vending machine example can be characterised by events (*coin*, *coffee*, and *tea*). Sequences of these events form traces of the system, which can be used as observations. Then, the predicate for the full system is $S_1(tr, tr') = (tr' = tr^\frown\langle tea\rangle \vee tr' = tr^\frown\langle coffee\rangle)$. Note that the observations before and after execution are traces. The trace afterwards (tr') is a concatenation of the trace before (tr) with the executed event. Analogously, $S_2(tr, tr') = (tr' = tr^\frown\langle tea\rangle)$. Again, we can show that S_1 is refined by S_2, i.e., $S_1 \sqsubseteq S_2$:

$$
\begin{aligned}
& S_1 \sqsubseteq S_2 && \{\text{Definition 3.11}\}\\
&= \forall \overline{v}, \overline{v}' : (S_2(\overline{v}, \overline{v}') \Rightarrow S_1(\overline{v}, \overline{v}')) && \{\overline{v} = \langle tr\rangle\} \text{ and } \{\overline{v}' = \langle tr'\rangle\}\\
&= \forall tr, tr' : (S_2(tr, tr') \Rightarrow S_1(tr, tr'))\\
& \qquad \{S_1(tr, tr') = (tr' = tr^\frown\langle tea\rangle \vee tr' = tr^\frown\langle coffee\rangle)\} \text{ and } \{S_2(tr, tr') = (tr' = tr^\frown\langle tea\rangle)\}\\
&= \forall tr, tr' : ((tr' = tr^\frown\langle tea\rangle) \Rightarrow (tr' = tr^\frown\langle tea\rangle \vee tr' = tr^\frown\langle coffee\rangle))\\
& && \{\text{first-order predicate calculus}\}\\
&= \forall tr, tr' : (\neg(tr' = tr^\frown\langle tea\rangle) \vee (tr' = tr^\frown\langle tea\rangle \vee tr' = tr^\frown\langle coffee\rangle))\\
& && \{\text{first-order predicate calculus}\}\\
&= \forall tr, tr' : (true \vee tr' = tr^\frown\langle coffee\rangle) && \{\text{first-order predicate calculus}\}\\
&= \forall tr, tr' : true && \{\text{first-order predicate calculus}\}\\
&= true
\end{aligned}
$$

Again, the other direction does not hold: S_2 is not refined by S_1, i.e., $\neg(S_2 \sqsubseteq S_1)$. □

As will be seen in Chapter 6, our notion of refinement will be based on both kinds of observations, i.e., on states and also on traces observed before and after execution.

Refinement relations do not distinguish between the inputs and outputs of a system. In the following section, we present the Input-Output Conformance (ioco) relation, which treats inputs and outputs in a different way.

3.2 Input-Output Conformance

The ioco relation was originally presented by Tretmans [191]. Informally, an implementation is input-output conform to a specification if it does not show output that is not specified. For unspecified inputs, the system may behave arbitrarily. The ioco relation is defined over LTSs with inputs and outputs, i.e., on an operational semantics. It can be used for all modelling formalisms with semantics that can be described via LTSs with inputs and outputs. For example, the process description language LOTOS [44] has a formal LTS semantics. Furthermore, the semantics of Symbolic Transition Systems (STSs) is defined in terms of LTSs. The following descriptions are mainly based on papers by Tretmans [191, 192].

Definition 3.12 (Labelled Transition System (LTS) with Inputs and Outputs)
An LTS M is tuple $M = \langle S, (L \cup \{\tau\}), T, s_0\rangle$ where

- S is a finite set of states,

- $L = L_I \cup L_O$ is a finite alphabet, i.e., a set of labels, partitioned into an input alphabet L_I and an output alphabet L_O with $L_I \cap L_O = \emptyset$,
- $\tau \notin L$ denotes an unobservable, internal action,
- $T \subset S \times (L \cup \{\tau\}) \times S$ is the transition relation, and
- $s_0 \in S$ is the initial state.

We refer to the class of LTSs with inputs and outputs as $LTS(L_I, L_O)$ in the following. Furthermore, we use the following common notations.

Definition 3.13 (LTS Notation)
Given an LTS $M = \langle S, (L \cup \{\tau\}), T, s_0 \rangle$ and let $s, s', s_1 \ldots, s_n \in S$, and let $S' \subseteq S$, and let $a, a_1, \ldots, a_n \in L$, and let $\sigma \in L^*$, then

$$
\begin{aligned}
s \xrightarrow{a} s' &=_{df} (s, a, s') \in T && (3.1)\\
s \xrightarrow{a} &=_{df} \exists s' : (s, a, s') \in T && (3.2)\\
s \stackrel{a}{\not\rightarrow} &=_{df} \nexists s' : (s, a, s') \in T && (3.3)\\
s \overset{\epsilon}{\Rightarrow} s' &=_{df} s = s' \vee \exists s_1, \ldots, s_n : s = s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{\tau} s_n = s' && (3.4)\\
s \overset{a}{\Rightarrow} s' &=_{df} \exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s' && (3.5)\\
s \overset{\sigma}{\Rightarrow} s' &=_{df} \exists s_1, \ldots, s_{n+1} : s = s_1 \overset{a_1}{\Rightarrow} s_2 \overset{a_2}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_{n+1} = s' \\
&\qquad \text{with} \quad \sigma = \langle a_1, \ldots, a_n \rangle \quad \text{or} \quad \sigma = \epsilon = \langle\,\rangle && (3.6)\\
s \overset{\sigma}{\Rightarrow} &=_{df} \exists s' : s \overset{\sigma}{\Rightarrow} s' && (3.7)\\
s \textit{ after } \sigma &=_{df} \{s' \mid s \overset{\sigma}{\Rightarrow} s'\} && (3.8)\\
M \textit{ after } \sigma &=_{df} s_0 \textit{ after } \sigma && (3.9)\\
\textit{traces}(s) &=_{df} \{\sigma \mid s \overset{\sigma}{\Rightarrow}\} && (3.10)\\
\textit{traces}(M) &=_{df} \textit{traces}(s_0) && (3.11)
\end{aligned}
$$

Informally, these notations can be described as follows. Equation 3.1 introduces $s \xrightarrow{a} s'$ as an alternative notation for the transition (s, a, s'). Then, $s \xrightarrow{a}$ states that there starts a transition labeled by a from state s leading to some successor state (Equation 3.2). Equation 3.3 states the contrary, i.e., that there does not exist such a transition. $s \overset{\epsilon}{\Rightarrow} s'$ means that either both states are the same ($s = s'$) or that there exists a set of intermediate states such that s' is reachable from s via a sequence of internal (τ) transitions only (Equation 3.4). $s \overset{a}{\Rightarrow} s'$ means that it is possible to reach state s' from state s by action a, while internal transitions may be needed before or afterwards (Equation 3.5). Equation 3.6 states that state s' is reachable from state s by a trace σ. If σ is a non-empty sequence of transitions labeled with $\langle a_1, \ldots, a_n \rangle$, internal actions may be required before, after, or between the transitions. If σ is the empty trace, then either s and s' coincide or there are only τ transitions between the states. Equation 3.7 defines that there exists a state s' that is reachable from s by trace σ. The relation *after* (Equation 3.8) determines the set of states reachable after trace σ starting from state s. For an LTS M, this set is defined beginning at the initial state s_0 (Equation 3.9). The set of traces of a state s is defined as all possible sequences of events starting at this state without internal events (Equation 3.10). The set of traces of an LTS M is defined by all traces starting from its initial state (Equation 3.11).

Definition 3.14 (Finite LTS)
An LTS $M = \langle S, (L \cup \{\tau\}), T, s_0 \rangle$ has finite behaviour if all of its traces have finite length, i.e., if there exists a natural number n such that all traces in $\textit{traces}(M)$ have a length smaller than n.

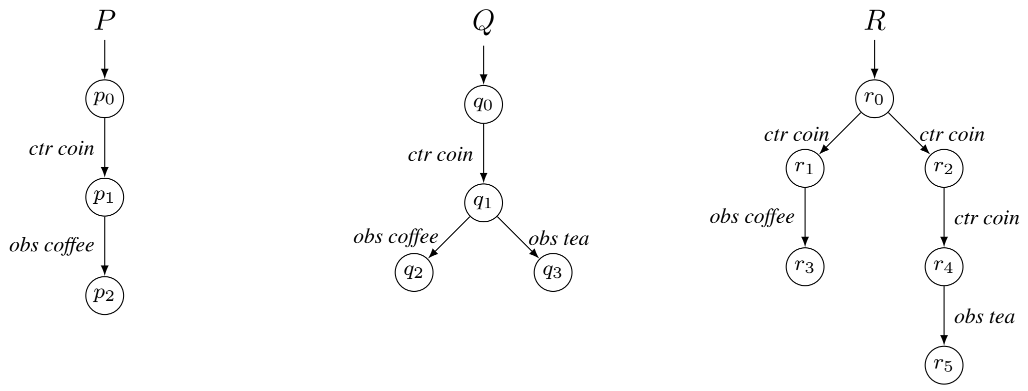


Figure 3.1: Examples for LTSs with inputs and outputs. Input actions are identified by prefix *ctr*, output actions are prefixed by *obs*.

Definition 3.15 (Deterministic LTS)
An LTS $M = \langle S, (L \cup \{\tau\}), T, s_0 \rangle$ is deterministic if $\forall \sigma \in L^* : |s_0 \; \mathit{after} \; \sigma| \leq 1$ holds.

Example 3.3. Figure 3.1 shows three LTSs with inputs and outputs: P, Q, and R. They describe variations of vending machines that deliver coffee and tea after insertion of one or several coins. The LTSs are represented as graphs. Nodes represent the states and labelled edges represent the transitions. The initial state is marked by an incoming arrow. For example, Q represents the LTS $\langle S, (L_I \cup L_O), T, s_0 \rangle$ with the following components:

- $S = \{q_0, q_1, q_2, q_3\}$
- $L_I = \{\mathit{coin}\}$ and $L_O = \{\mathit{coffee}, \mathit{tea}\}$
- $T = \{(q_0, \mathit{coin}, q_1), (q_1, \mathit{coffee}, q_2), (q_1, \mathit{tea}, q_3)\}$
- $s_0 = q_0$

The transition relation T can alternatively be written as $\{q_0 \xrightarrow{\mathit{coin}} q_1, q_1 \xrightarrow{\mathit{coffee}} q_2, q_1 \xrightarrow{\mathit{tea}} q_3\}$. It holds that $q_0 \xrightarrow{\mathit{coin}}$, but $q_0 \overset{\mathit{tea}}{\not\rightarrow}$. Furthermore, $q_0 \overset{\mathit{coin}}{\Longrightarrow}$, $q_0 \overset{\mathit{coin},\mathit{coffee}}{\Longrightarrow}$, and $q_0 \overset{\mathit{coin},\mathit{tea}}{\Longrightarrow}$. Hence, the traces of LTS Q are $\mathit{traces}(Q) = \{\langle\rangle, \langle \mathit{coin} \rangle, \langle \mathit{coin}, \mathit{coffee} \rangle, \langle \mathit{coin}, \mathit{tea} \rangle\}$. The set of states that can be reached from state q_0 following trace $\langle \mathit{coin}, \mathit{tea} \rangle$ is a singleton: $q_0 \; \mathit{after} \; \langle \mathit{coin}, \mathit{tea} \rangle = \{q_3\}$. □

A state s from which the system cannot proceed without additional inputs from the environment is called *quiescent*, denoted as $\delta(s)$. In such a state, all output and internal events are disabled.

Definition 3.16 (Quiescent State)
Given an LTS $M = \langle S, (L_I \cup L_O \cup \{\tau\}), T, s_0 \rangle$. A state $s \in S$ is quiescent, denoted as $\delta(s)$, iff $\forall a \in (L_O \cup \{\tau\}) : s \overset{a}{\not\rightarrow}$.

For observing quiescence, the transition relation T is extended by adding self-loops with the special label δ at quiescent states: $T_\delta =_{\mathit{df}} T \cup \{(s, \delta, s) \mid s \in S \wedge \delta(s)\}$. Let M_δ be the LTS over the alphabet $L \cup \{\tau, \delta\}$ resulting from adding δ self-loops to an LTS M.

Example 3.4. Figure 3.2 shows the three LTSs P_δ, Q_δ, and R_δ that result from the LTSs P, Q, and R depicted in Figure 3.1 by adding quiescence. The transitions representing quiescence are self-loops labelled by δ and have been added in each quiescent state. They are highlighted in blue. □

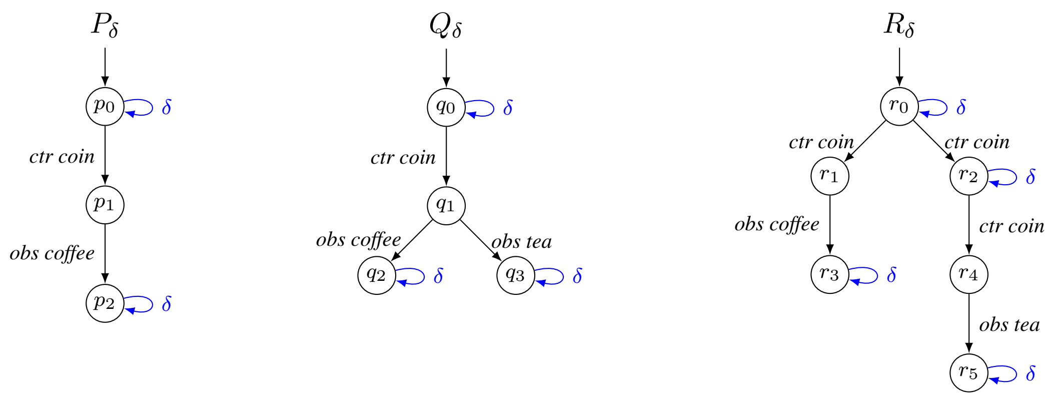


Figure 3.2: These three LTSs were created from the LTSs in Figure 3.1 by adding quiescence (δ).

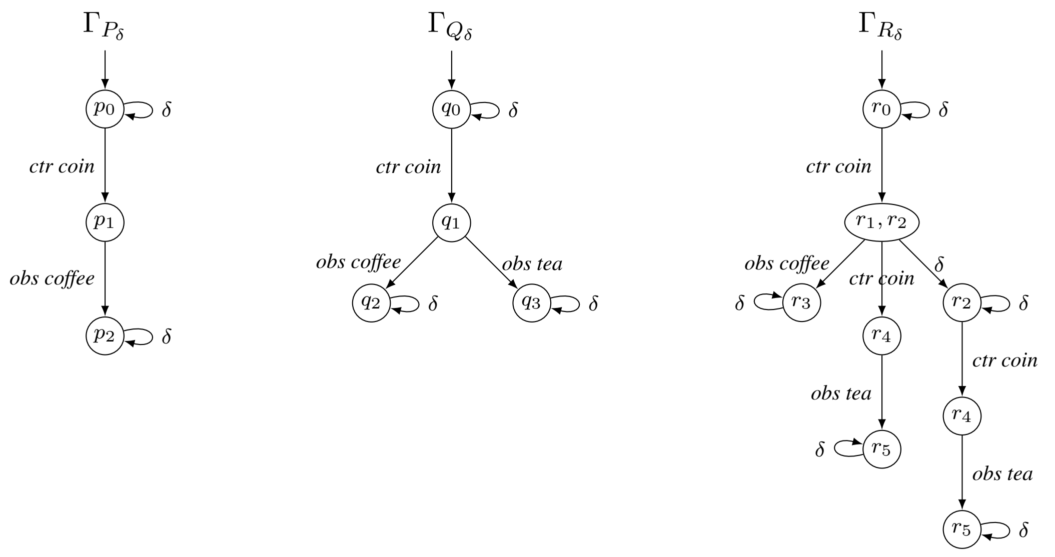


Figure 3.3: These three LTSs represent the suspension automata of the LTSs P_δ, Q_δ, and R_δ in Figure 3.2. Note that $P_\delta = \Gamma_{P_\delta}$ and $Q_\delta = \Gamma_{Q_\delta}$.

The deterministic automaton obtained via subset construction [120] from M_δ is called *suspension automaton* Γ [191]. Its behaviour is defined via the set of suspension traces of M_δ:.

Definition 3.17 (Suspension Traces)
Given an LTS $M_\delta = \langle S, (L \cup \{\tau, \delta\}), T_\delta, s_0\rangle$, then its suspension traces are defined as:

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

Example 3.5. Figure 3.3 shows the suspension automata Γ_{P_δ}, Γ_{Q_δ}, and Γ_{R_δ} of the LTSs P_δ, Q_δ, and R_δ in Figure 3.2. Note that $P_\delta = \Gamma_{P_\delta}$ and $Q_\delta = \Gamma_{Q_\delta}$ as those LTSs were already deterministic. Γ_{R_δ} is obtained by determinising R_δ via subset construction. $\square$

The set of outputs that are enabled in a state, or in a set of states respectively, is defined as follows:

Definition 3.18 (Outputs of States)
Given an LTS $M = \langle S, (L_I \cup L_O \cup \{\tau\}), T, s_0\rangle$ and $s \in S$ and $S' \subseteq S$, then

$$out(s) =_{\mathit{df}} \{a \in L_O \mid s \overset{a}{\rightarrow}\} \cup \{\delta \mid \delta(s)\} \qquad \text{and} \qquad out(S') =_{\mathit{df}} \bigcup_{s \in S'} out(s)$$

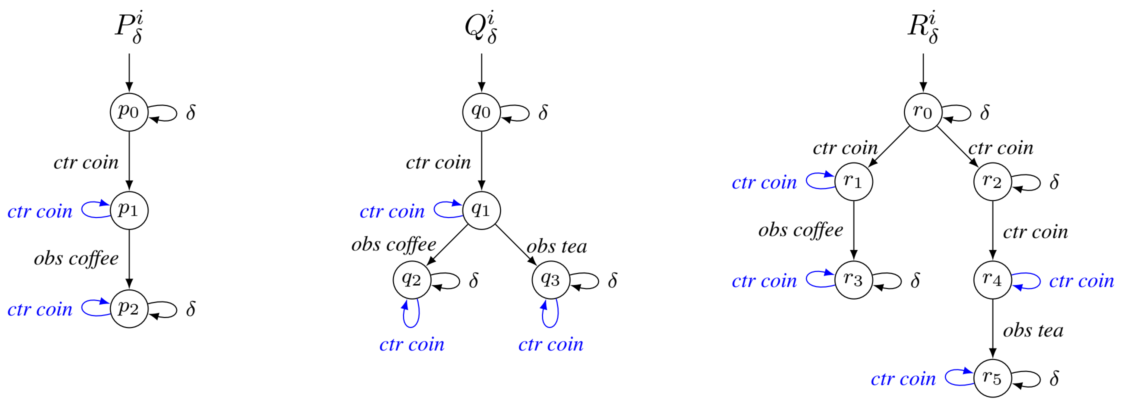


Figure 3.4: Examples for Input Output Transition Systems (IOTSs).

Example 3.6. Consider again the LTS Q_δ of Figure 3.2. For example, $out(q_0) = \{\delta\}$ and $out(q_1) = \{coffee, tea\}$. □

For the ioco relation, SUTs are considered to be weakly input-enabled, i.e., all inputs (possibly preceded by τ-transitions) are enabled in all states. This class of LTSs is referred to as $IOTS(L_I, L_O)$:

Definition 3.19 (Input Output Transition System (IOTS))
An input output transition system is an LTS $M = \langle S, (L_I \cup L_O \cup \{\tau\}), T, s_0 \rangle$ with the following property: $\forall a \in L_I, \forall s \in S : s \overset{a}{\Rightarrow}$

Hence, IOTSs are a subclass of LTSs, i.e., $IOTS(L_I, L_O) \subseteq LTS(L_I, L_O)$.

Example 3.7. Figure 3.4 shows three IOTSs. The used input alphabet has only one element: *coin*. It is enabled in each state. The IOTSs P_δ^i, Q_δ^i, and R_δ^i have been created from the LTSs in Figure 3.2 by adding self-loop transitions labelled by inputs that are not enabled in a state. This way of making an LTS input-enabled is called *angelic completion*. The basic idea behind it is that unknown inputs are always accepted, but ignored. The transitions added during angelic completion are highlighted in blue. □

The ioco relation states that for all suspension traces of 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. Formally, we have the following definition.

Definition 3.20 (Input Output Conformance (ioco))
Given an implementation model $I \in IOTS(L_I, L_O)$ and a specification $S \in LTS(L_I, L_O)$, then the relation ioco is defined as:

$$I \ ioco \ S \quad =_{df} \ \forall \sigma \in Straces(S) : out(I \ after \ \sigma) \subseteq out(S \ after \ \sigma)$$

Example 3.8. Consider the IOTSs in Figure 3.4 as implementations. The suspension automata in Figure 3.3 serve as specifications. We have that $P_\delta^i \ ioco \ \Gamma_{P_\delta}$, $Q_\delta^i \ ioco \ \Gamma_{Q_\delta}$, and $R_\delta^i \ ioco \ \Gamma_{R_\delta}$. We have also that $P_\delta^i \ ioco \ \Gamma_{Q_\delta}$. However, $\neg(Q_\delta^i \ ioco \ \Gamma_{P_\delta})$ due to the unspecified output *tea* after the *coin* input. Furthermore, $\neg(R_\delta^i \ ioco \ \Gamma_{P_\delta})$ as the specification does not include quiescence after the *coin* input. However, considering an implementation I as depicted in Figure 3.5, then $I \ ioco \ \Gamma_{P_\delta}$ holds, as *tea* is only delivered after the input *button*, which is not a trace of the specification. In this case, underspecification in Γ_{P_δ} allows for arbitrary behaviour in I after unspecified inputs. □

Additionally to refinement (Definition 3.11), we also use ioco in this work. The ioco formalisation of Tretmans [191] is based on an operational semantics. In contrast, Weiglhofer and Aichernig [199] used

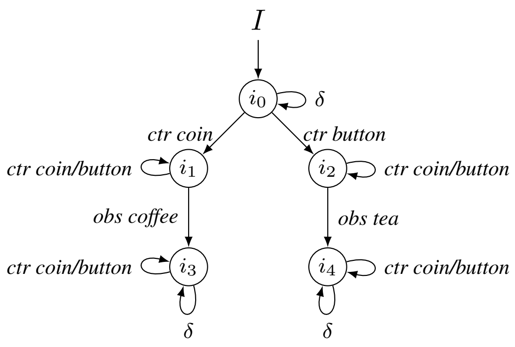


Figure 3.5: Example implementation.

the denotational predicative semantics of UTP [119] to define ioco. In this way, refinement and ioco are both defined using the same theory and can be related directly. Weiglhofer and Aichernig proved that refinement implies ioco, i.e., if an implementation refines a specification, it is also ioco-conform to the specification. Note that the other direction does not necessarily hold.

One of the main advantages of ioco is that it allows for incomplete (partial) specifications since it is only defined over traces of the specification. For unspecified inputs, the system may behave arbitrarily. This concept is similar to pre-/post-conditions, where post-conditions do not have to be fulfilled if the according pre-condition is violated. Partial modelling is important in practice. Often, the full behaviour of complex systems cannot be specified in one monolithic model. Instead several partial models, each focusing on different aspects of the system, can be created.

Many variations of ioco exist. For timed systems, different notions of input output conformance have been presented: the relativised timed input output conformance (*rtioco*) is used by the UPPAAL tools [112, 147]. Furthermore, *tioco* [141] and $\sqsubseteq_{tioco}$ [54] have been defined. In contrast to *rtioco* and *tioco*, the latter considers quiescence. For a more detailed relation of the three timed conformance relations, we refer to Krichen and Tripakis [142]. As the assumption of input-enabledness of implementations is too strong for many SUTs, Lestiennes and Gaudel developed the *rioco* relation, which is focused on testing systems that are not input-enabled [151, 150]. For hybrid systems, *hioco* has been presented [196]. A symbolic variant of ioco, named *sioco*, has been defined over symbolic transition systems (STSs) [93].

Furthermore, Hierons et al. [116] identified problems with the ioco relation in testing of distributed systems. If the SUT has physically distributed interfaces, so-called ports, then a tester is placed at each port. Moreover, it is assumed that the individual testers cannot communicate with each other and that there is no global clock available. In such a setting, ioco may lead to incorrect verdicts. Hierons et al. defined three ioco variants. The first one is *p-dioco* and corresponds to the setting described above. The second relation is called *dioco*. It is designed for situations, when there is a central agent that can gather all of the information from the individual agents. Finally, *c-dioco* has been developed for use with controllable test cases only. Recently, the *dioco* relation has been adapted for timed, distributed systems resulting in the *dtioco* relation [98]. Another conformance relation (named *mioco*) for systems with multiple ports has been presented by Brinksma et al. [53]. However, it assumes only one central tester that controls and observes all of the ports.

3.2.1 Assumptions of ioco

The ioco relation relies on several assumptions, which were formalised by Weiglhofer and Aichernig [199]. Some of them have already been mentioned above. In the following, we summarise all assumptions underlying ioco.

No Divergence (Livelocks): The ioco relation does not consider livelocks, i.e., divergence, which denotes infinite sequences of internal actions. In the ioco theory, implementations, specifications, and hence also the test cases, which are derived from the specifications, must not contain livelocks. Note that there exist conformance relations that can handle livelocks, e.g., the failures divergences refinement relation already described in Section 3.1.3.

Observability of Quiescence: In order to check for ioco, the absence of outputs or internal actions of the implementation, i.e., quiescence, has to be observable in specifications as well as in implementations. The following assumptions concern only the implementation of the SUT.

Distinction between Internal and External Choices: Implementations that can be checked for ioco conformance have to adhere to the following rules for choices over inputs and outputs:

- For implementations, choices over its outputs are *internal choices*, i.e., it is up to the SUT to decide which output it wants to send.
- In contrast, choices over inputs to the SUT are *external choices* from the implementation's point of view. The environment may choose which input will be sent to the SUT.
- Furthermore, an implementation may be in a state where inputs and outputs are enabled at the same time. In this case, it is up to the environment to decide if it either sends an input to the SUT or if it accepts outputs from the SUT. In other words, choices over inputs and outputs are *external choices*. As a consequence, the environment may prevent the SUT to provide an output.

For the latter two items, i.e., choices over sets of actions that contain inputs, the following property of an implementation is essential.

Input-Enabledness: The ioco relation only works for implementations that always accept every input out of their alphabet. Such implementations are denoted as *input-enabled*.

Fairness: The *fairness assumption* affects non-deterministic implementations. It requires that an implementation eventually shows all of its possible non-deterministic behaviours provided that it is re-executed often enough. Without this fairness assumption, an incorrect, non-deterministic implementation could always provide only the correct, but never the incorrect non-deterministic behaviour in response to a test case. Hence, the erroneous implementation would have to be classified correct.

3.2.2 Other Conformance Relations for Labelled Transition Systems

The ioco relation defined above (Definition 3.20) can be varied in the set of traces that has to be considered. In this way, a generic ioco relation can be defined [192]:

Definition 3.21 (Generic Input Output Conformance)
Given an input alphabet L_I and an output alphabet L_O then $ioco_{\mathcal{F}} \subseteq IOTS(L_I, L_O) \times LTS(L_I, L_O)$ is defined as:

$$I \; ioco_{\mathcal{F}} \; S \;\; =_{df} \;\; \forall \sigma \in \mathcal{F} : out(I \; after \; \sigma) \subseteq out(S \; after \; \sigma)$$

For ioco, $ioco_{\mathcal{F}}$ is instantiated by $\mathcal{F} = Straces(S)$. Using different sets of event sequences for $\mathcal{F}$, various other conformance relations can be defined:

Definition 3.22 (Input Output Testing Relation)
The input output testing relation $\leq_{iot}$ can be defined by instantiating $ioco_{\mathcal{F}}$ with $\mathcal{F} = L^*$.

Definition 3.23 (Input Output Refusal Relation)
The input output refusal relation $\leq_{ior}$ can be defined by instantiating $ioco_{\mathcal{F}}$ with $\mathcal{F} = (L \cup \{\delta\})^*$.

Definition 3.24 (ioconf)
The conformance relation *ioconf* can be defined by instantiating $ioco_{\mathcal{F}}$ with $\mathcal{F} = traces(S)$.

Note that the input output testing relation and ioconf do not consider quiescence. The input output refusal relation already knows about quiescence, but is not restricted to sequences of events of the specification. Hence, it cannot support partial specifications like ioco. For a detailed comparison of the defined conformance relations and their strengths and weaknesses, we refer to Tretmans [191, 192].

All of the above defined conformance relations for LTSs distinguish between inputs and outputs. In the following, we briefly present the most relevant conformance relations without this distinction [191, 113]. *Trace preorder* is given if the traces of the implementation I are included in the traces of the specification S, i.e., $traces(I) \subseteq traces(S)$. Note that trace preorder corresponds to traces refinement (cf. Section 3.1.2). Sometimes, it is also called trace inclusion. *Testing preorders* differentiate conformance between two systems not only by their normal traces, but also by their complete traces, i.e., deadlocks. A further conformance relation is called *conf*. It weakens testing preorder by considering only specified behaviour. Hence, *conf* checks whether the implementation does not show unspecified deadlocks or traces, but does not check for extra behaviour of the implementation. Finally, *refusal preorder* considers so-called *failure traces*, which are traces including both the accepted and the refused actions. Refusal preorder resembles failures refinement (cf. Section 3.1.3).

3.3 Classifying Conformance Relations

Conformance relations can be divided into two categories: those checking *global* properties and others relying on *local* properties [21]. Global properties denote sequences of observations, i.e., all kinds of traces. Global conformance relations include the already mentioned traces refinement or trace preorder, failures refinement, as well as ioco. Local properties are bound to specific states, e.g., observations enabled in a specific state or the state information itself. Local conformance relations include all kinds of simulation preorders, for example alternating simulation [21]. Alternating simulation is a relation between the states of two systems and hence a local conformance relation. Informally, it requires that the implementation can only make outputs that the specification allows, and the specification can only make inputs that the implementation can also make. This has to hold for all related states. In this case, the implementation is not considered to be input-enabled. Hence, the specification is restricted to give only inputs that can be processed by the implementation.

Generally, checking for global conformance relations is computationally more expensive than checking for local conformance. Considering global properties, all non-deterministic choices along a specified trace have to be considered. For local conformance checks, it is sufficient to consider only the next events enabled in a certain state. However, local conformance relations do not interpret non-determinism as underspecification. They are very strict and check whether the local choices of the SUT correspond to the local choices in the specification. In this way, implementations may be rejected due to wrong internal choices. A possibility to counteract would be to determinise beforehand. Considering only deterministic systems, some local conformance relations coincide with global conformance relations. For example, for deterministic systems, ioco and alternating simulation are equivalent [197]. However, in black-box testing determinisation of the SUT is no option since we do not have access to its source code.

4 Model-Based Mutation Testing

Parts of this chapter are based on the author's master's thesis [131], which already dealt with model-based testing. Additionally, some parts have been published in MBT 2012 [14], CSTVA 2012 [15], QSIC 2012 [13], and TAP 2013 [16].

This chapter presents model-based testing as well as mutation testing and explains a combination thereof called model-based mutation testing.

4.1 Model-Based Testing

There are various definitions of Model-Based Testing (MBT) (sometimes also called specification-based testing) in the literature. All of them involve the use of a model for testing, whereas the modelled subject and the aim of MBT can differ. In the following, a few perceptions of MBT will be presented. Of course, this list cannot be exhaustive.

Frantzen et al. [93] describe MBT as a black-box testing technique. The goal of MBT is to test whether a SUT conforms to a formal specification (model) of the SUT. The model can be used for automatic test case generation and as an oracle (cf. Definition 2.16), i.e., test result evaluation can be performed automatically, which requires a formal conformance relation (cf. Chapter 3).

Utting and Legeard [194] relate MBT very closely to *automation*. They focus on MBT in terms of generating test cases with oracles based on behavioural models. MBT covers the generation of (a) input values, (b) call sequences and (c) oracles for checking the test results, whereas the automating aspect is emphasised. The authors define MBT as *"the automation of the design of black-box tests"*.

Similarly, Pretschner and Philipps [172] describe the main idea of model-based testing as the use of models to express the intended behaviour of a system. These models are then used to derive test cases including input and expected output, which can be run on the SUT. Unlike the two already introduced approaches, the authors do not identify automation as a characteristic of MBT. The manual derivation of test cases from a model also belongs to MBT.

Binder [40] claims that testing should always be model-based. Testing can be seen as searching for bugs. Exhaustive testing is infeasible. Hence, testing has to be systematic, focused, and automated. MBT has all of these three attributes.

Three further approaches, which are sometimes interpreted as MBT, are mentioned by Utting and Legeard [194]:

- The generation of test input data from a domain model.
- The generation of test cases from a model of the environment of the SUT.
- The generation of test scripts from abstract tests.

This work does not correspond to these three approaches, but deals with MBT as defined above:

Definition 4.1 (Model-Based Testing)
Model-based testing (MBT) denotes the generation of test cases including oracles based on behavioural models.

4.1.1 The Model-Based Testing Process

According to Utting and Legeard [194], the process used in MBT consists of the following five steps:

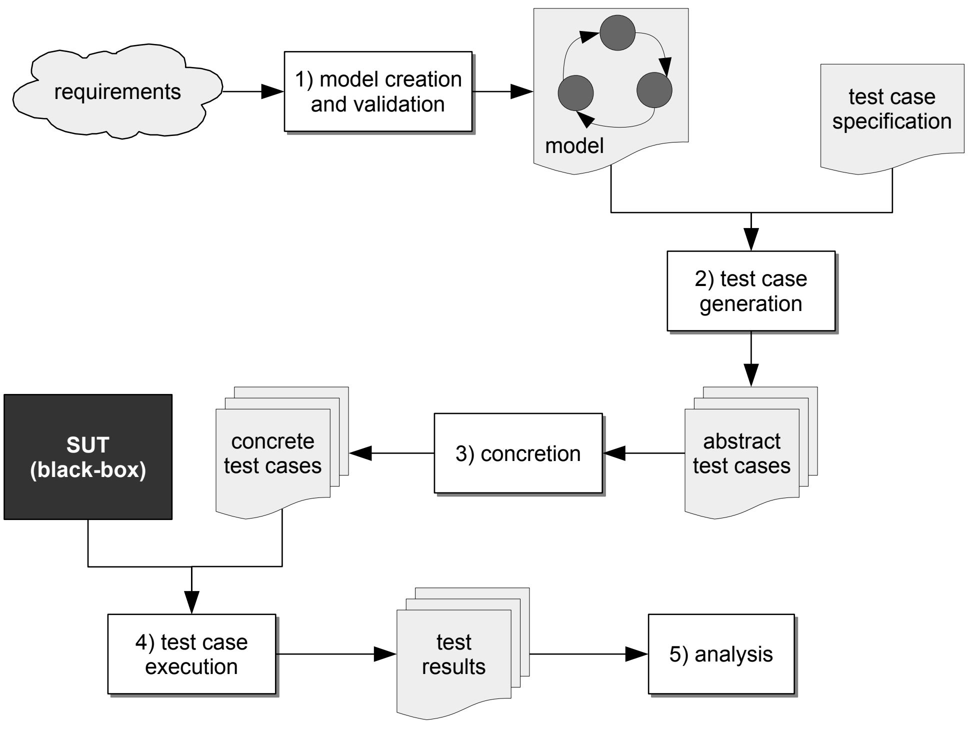


Figure 4.1: The model-based testing process: Based on the requirements, a formal model is created and validated. The model and a test case specification are used for the automatic generation of abstract test cases. After concretion of the abstract test cases, the tests are executed on the SUT and verdicts are assigned. Finally, the test results are analysed.

1. Model creation of the SUT and/or its environment.
2. Generation of abstract test cases from the model.
3. Generation of concrete, executable test cases by concretion of the abstract test cases.
4. Execution of the concrete test cases on the SUT and assignment of verdicts.
5. Analysis of the test results.

Figure 4.1 was influenced by Figure 2.4 of Utting and Legeard [194] and Figure 10.1 of Pretschner and Philipps [172]. It depicts the MBT process as defined above and completes it by inserting model validation, which should accompany the creation of the model. Additionally, a test case specification was introduced, which serves as selection criterion [172].

Hence, the process of model-based testing is the following [194]:

1. Model Creation and Validation Based on the requirements, the system and/or its environment has to be modelled. According to Stachowiak [185], a model has the following three properties:

- A model is a *mapping* from a concrete (“original”) into a more abstract (“model”) world.
- A model serves a specific *purpose*.
- A model is a *simplification*. It does not reflect all attributes of the concrete world.

Models created in this step are referred to as test models:

Definition 4.2 (Test Model)
A test model is a formal model that describes the expected behaviour of the SUT. It shows all of the above defined properties of a model. It is much simpler and smaller than the SUT, it should be focused on the aspects of the SUT that shall be tested, and details should be abstracted.

Since the test model will be used for verification of the SUT, it has to be validated. It is necessary to check whether the test model correctly represents the user requirements.

2. Test Case Generation The created model is now used to automatically generate test cases, which are sequences of operations of the model (cf. Definition 2.13). To define which tests shall be generated from the model, a test case specification is necessary. Without such a specification, usually an unlimited number of test cases could be generated. For automatic test case generation, different algorithms and heuristics are used. The resulting test cases are abstract:

Definition 4.3 (Abstract Test Case)
Since the model is a simplification of the SUT, the resulting test cases are not detailed enough to be directly executable. For example, parameters may be uninstantiated.

3. Concretion To make the abstract test cases executable, they have to be concretised. The goal of this step is to close the gap between the abstract test cases and the concrete SUT. Figure 4.2 is based on a figure by Pretschner and Philipps [172] and illustrates how concretion and abstraction are used to connect the “model” world with the “original” world. The abstract input i, which is defined for the “model” world, has to be transferred into the “original” world. In order to be a valid input for the SUT, the abstract input i has to be concretised via the concretion function γ. The resulting test cases are called concrete test cases:

Definition 4.4 (Concrete Test Case)
Concrete test cases are on the same level of abstraction as the SUT and are directly executable on the SUT.

4. Test Case Execution and Assignment of Verdicts By now, the concrete test cases can be executed. Again, Figure 4.2 will be used for illustration. The SUT processes the concrete input $\gamma(i)$ and produces some concrete output o'. When executing a test case, the actual output of an SUT has to be compared to the expected output to assign verdicts for each test (cf. Definition 2.19). In MBT, the test model serves as an oracle (cf. Definition 2.16), i.e., the expected output o can be derived from the model. This means that the expected output is defined in the “model” world. Hence, the concrete output from the SUT has to be abstracted via the abstraction function α in order to be compared to the abstract expected output o to generate a verdict.

5. Analysis of the Test Results Finally, the results of the test executions have to be analysed. For each test reporting a failure, the fault causing this failure has to be found. This fault is not necessarily located in the SUT. It could lie within the implementation of the concretion function γ. It could also be in the test case, which means that it would be in the model used for test case generation. Thus, the analysis of the test results also helps validating the model [36, 194].

This work mainly focuses on the generation of test cases (Step 2). However, Chapter 11 also covers the remaining steps including concretion, and execution of the abstract test cases as well as the analysis of the results.

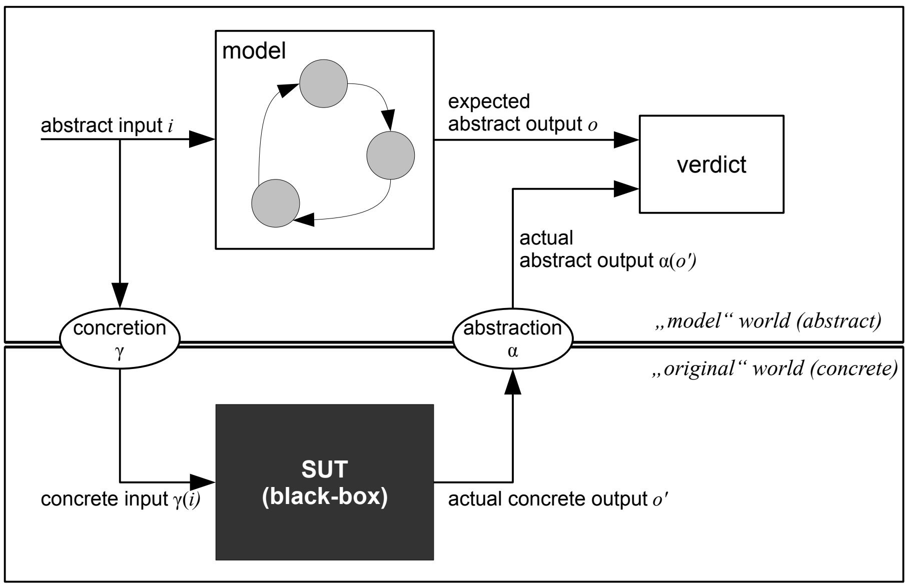

model
abstract input i
expected abstract output o
verdict
actual abstract output $\alpha(o')$
concretion γ
abstraction α
„model“ world (abstract)
„original“ world (concrete)
SUT (black-box)
concrete input $\gamma(i)$
actual concrete output o'

Figure 4.2: Abstraction and concretion are needed in model-based testing to connect the “model” world with the “original” world.

4.1.2 Benefits and Limitations of Model-Based Testing

According to Utting and Legeard [194], MBT achieves good results. In several case studies, the model-based testing approach found the same number of errors in the SUT or even more compared to manually designed tests. The authors also state that the quality of model-based test cases is better than the quality of manually designed tests. Manual testing requires an experienced tester, who is good at guessing sources of error. In addition, the manual test design process is often not reproducible. Since MBT uses algorithms and heuristics to automatically generate test cases, the test design is more systematic and reproducible. Due to automation, it is possible to produce a great number of test cases, which can help to find more bugs.

The requirements for the SUT are typically informal and formulated in a natural language. Hence, they are possibly ambiguous, incomplete, or contradictory. When building a model from the requirements to describe the intended behaviour of the system, problems in the informal requirements can be revealed. Since the model is formal, which means it has precise semantics, encountered problems in the requirements have to be resolved. The clarification of requirements issues is crucial, because each resolved requirements problem means less faults during design and implementation. The earlier an error is found, the cheaper it is to fix. Frequently changing requirements necessitate the adaption of tests. The update of test suites written manually is time-consuming. With MBT, this task is easier. Only the model has to be updated and the tests can be generated anew [194].

According to Utting and Legeard [194], traceability, which is *“the ability to relate each test case to the model, to the test selection criteria, and even to the informal system requirements”*, is improved by MBT. For example, this can help to optimise test execution. In the case of changes to the model, only tests affected by changes need to be executed again.

Since models are a simplified version of the SUT, they are easier to understand, validate, and maintain than the SUT itself [36]. Particularly, models make it easier to automatically generate test cases. According to many scientists, e.g., Hierons et al. [113] or Tretmans [192], this is actually the main benefit of explicit model building and model-based testing.

All of the above described advantages of MBT help to reduce testing costs and time. Nevertheless, MBT has also drawbacks, which may outweigh the benefits and may prevent the cost and time balance to be positive. MBT involves an extra effort compared to conventional software testing: model building, validation, and maintenance. The analysis of the failed tests is also more complex, because there exist several sources of error: the SUT, the model, and the implementation of the concretion function [172, 194].

Although good results have been achieved with MBT, it is not the silver bullet to find software bugs. The abstraction and modelling skills of the tester as well as the selection of the test case specification have a great impact on the success of MBT. This may cause additional training costs when MBT is deployed the first time [194].

According to Utting and Legeard [194], the main field of application of MBT is functional testing. However, MBT can also be applied to non-functional requirements. For example, the UPPAAL tools COVER and TRON have been successfully used for MBT of real-time applications [112, 147].

The practicability of MBT also depends on the type of SUT. Not in every case, MBT is applicable. Sometimes, the deployment of manual testing is easier and leads to better results. Experience is required to decide whether MBT or conventional testing is more beneficial for testing a certain SUT. In order to apply MBT to generate appropriate test cases, the requirements and models have to be updated continuously. Otherwise, wrong properties of the SUT will be tested.

4.1.3 A Taxonomy of Model-Based Testing

Utting et al. [195] presented a taxonomy of model-based testing approaches that facilitates the comparison of different MBT tools and techniques. Figure 4.3 gives an overview of the taxonomy. The dimensions have been chosen according to the MBT process. *Model specification* obviously refers to the first step of model creation and validation. Here, the authors distinguish between the *scope* of the model, the model *characteristics*, and the modelling *paradigm* to classify an MBT approach. The *test generation* category addresses both *test selection criteria* (i.e., the choice of test specifications) and the *technology* used for test case generation. Finally, *test case execution* states the last dimension.

In the following, we classify the MBT approach used in this work according to this taxonomy. In Figure 4.3, we highlighted the characteristics of our approach using bold, blue font. Regarding the model scope, we clearly want to cover inputs as well as expected outputs that serve as an oracle. Our modelling formalism are action systems, which are described in detail in Chapter 5. In action systems, we support a limited notion of time. Furthermore, we allow for non-deterministic models. This is important for two reasons. First, non-determinism in models is required in order to support non-deterministic implementations. Second, good test models (Definition 4.2) are more abstract than the system under test. In this case, non-determinism is used in models to express implementation freedom (cf. operational refinement, Definition 3.9). Our models focus on discrete systems. Typical action systems are solely state-based. However, we customise them to add an event-based view, i.e., we cover both paradigms: state-based modelling (denoted *pre-post or input domains* in the taxonomy) and transition-based modelling. As already mentioned, we combine MBT and mutation testing for test case generation. Thus, our main test selection criterion is fault-based. However, our tool also supports the derivation of random test cases from the model. Hence, one used test generation technology is based on random generation. For fault-based test generation, we combine model-checking and constraint solving techniques. The resulting test cases are executed offline (cf. Definition 2.25).

For further literature on MBT, we refer to the surveys of Dias Neto et al. [80] and Hierons et al. [113]. Furthermore, Broy et al. [56] collected various articles on MBT. The book resulted from a research seminar at Schloss Dagstuhl in January 2004.

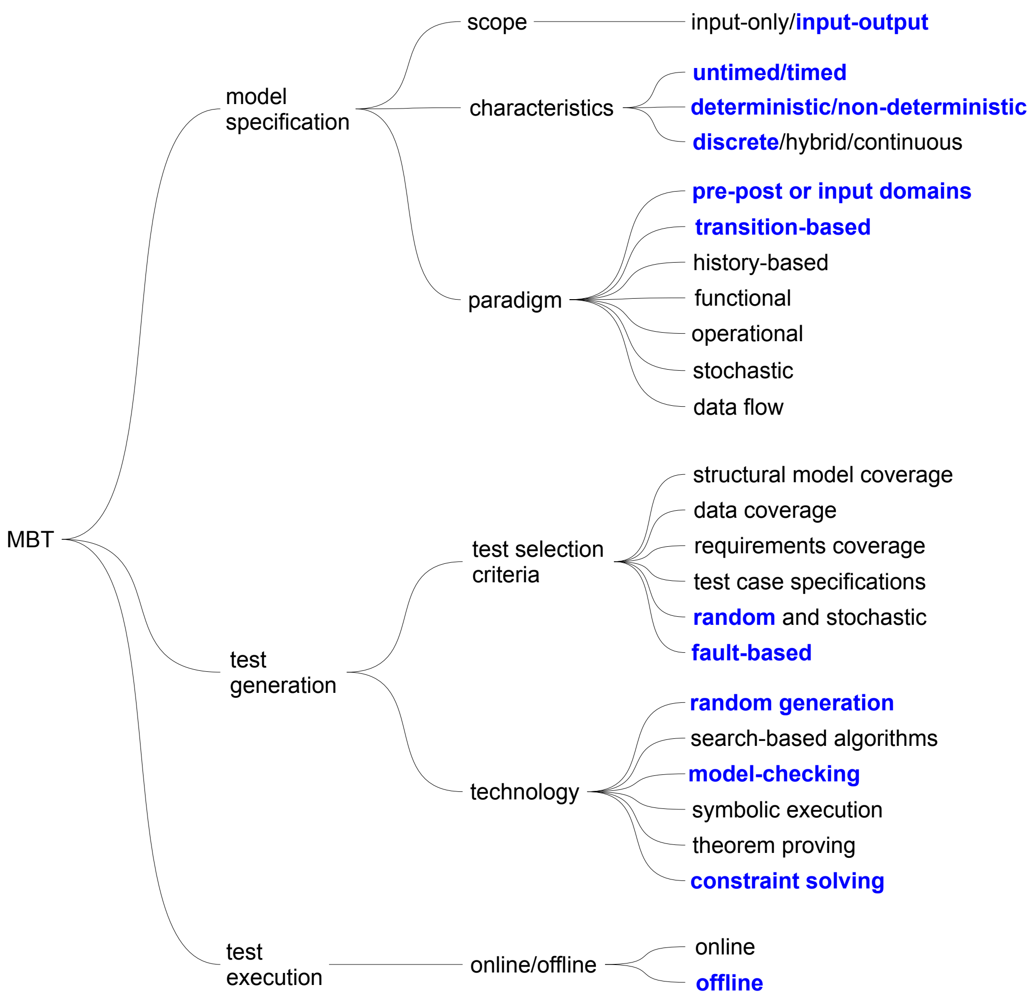


Figure 4.3: A taxonomy of model-based testing proposed by Utting et al. [195].

4.2 Mutation Testing

Mutation testing is a method to assess and increase the quality of a test suite. It was already introduced in the 1970s [109, 78]. Recently, Jia and Harman conducted a detailed survey on mutation testing [130]. Figure 4.4 illustrates the workflow of mutation testing [130]:

1. Run Test Suite on Original Program An original program P is tested given an initial test suite TS. If there are failing test cases, P has to be revised (Step 1a). This may need several iterations.

2. Mutation If the program P is correct according to the test suite TS, it is mutated. A set of mutation operators is applied on P resulting in a set of *mutants* P'.

Definition 4.5 (Mutation Operator)
Mutation operators are patterns of typical programming faults. They specify how to syntactically alter the source code of the original program P.

3. Run Test Suite on Mutants The test cases are then executed on the generated mutants. If a test case fails on a mutant, it *kills* the mutant. If a mutant cannot be killed by a test case in the test suite TS, it is a *live* mutant. If all mutants have been killed by the test suite, mutation testing is finished.

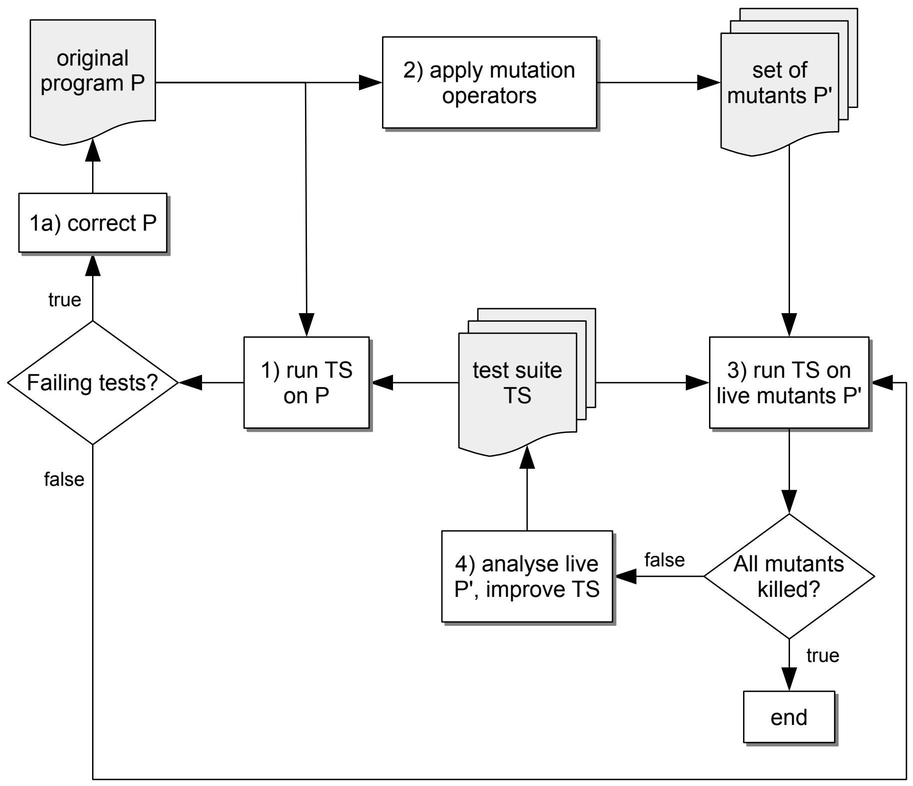


Figure 4.4: Mutation testing is an iterative process: An original program P is tested to be correct according to a given test suite TS. Afterwards, P is mutated resulting in a set of mutants P'. The test suite TS is then run on these mutants. If all mutants have been killed, mutation testing is finished. Otherwise, the live mutants are analysed, the test suite TS is improved if possible and the whole process is repeated.

4. Analyse Live Mutants and Improve Test Suite Otherwise, the remaining live mutants are analysed. Additional test cases that kill the remaining mutants are added to the test suite TS. The whole mutation testing process is iterative and aims at continuously improving TS. In the best case, all mutants get killed eventually.

Unfortunately, mutation testing is not that straightforward as not all mutants are faulty:

Definition 4.6 (Equivalent Mutant)
An equivalent mutant P' does not behave differently from the original program P. Consequently, it cannot be killed by any test case.

For example, the application of mutation operators on unreachable code has no effect and results in equivalent mutants. These mutants need to be identified by means other than testing. Traditionally, this is done by manual inspection, because program equivalence is undecidable in general. Jia and Harman [130] identified the equivalent mutants problem to be *"a barrier that prevents Mutation Testing from being more widely used"*.

Mutation testing relies on two assumptions that have been empirically confirmed [130]:

Definition 4.7 (Competent Programmer Hypothesis)
The competent programmer hypothesis states that programmers are skilled and do not write completely wrong implementations. It assumes that they only make small mistakes.

Definition 4.8 (Coupling Effect)
The coupling effect states that test cases that are able to detect simple failures (like the ones caused by mutation operators) are also able to reveal more complex bugs.

Additionally to test suite improvement, mutation testing can also be used for test suite assessment. The mutation score measures the effectiveness of a test suite in terms of its mutation detection ability. It is defined as the ratio of killed mutants to the number of non-equivalent mutants [130]:

Definition 4.9 (Mutation Score (MS))

$$MS = \frac{number\ of\ killed\ mutants}{total\ number\ of\ mutants - number\ of\ equivalent\ mutants}$$

Hence, the best possible value for MS is 1. In this case, all non-equivalent mutants are killed by the test suite. The worst value is 0, i.e., the test suite was not able to kill any mutant.

In addition to the equivalent mutants problem mentioned above, the potentially high runtimes for executing the large number of mutants against the test suite is a problem in mutation testing. Hence, one important field of research deals with reducing the number of mutants while preserving a high mutation score. A simple approach to reduce the number of mutants is *mutant sampling* [4, 60]. The full set of mutants is generated and a certain percentage of these mutants is subsequently selected for mutation testing. The spare mutants remain unused. A similar idea is used in *mutant clustering* [124, 128]. Here, the set of mutants is partitioned into clusters. Each cluster contains mutants that are killed by a similar set of test cases. A small set of mutants is selected from each cluster to be used in mutation testing. *Selective mutation* [157, 169] is also a technique to reduce the number of mutants to be processed. It avoids the generation of the full set of mutants by reducing the number of mutation operators. It is based on the observation that some mutation operators produce redundant mutations. Another direction of research to reduce the number of mutants investigated higher-order mutants [129]. Typically, first-order mutants are used in mutation testing.

Definition 4.10 (First-Order Mutant)
First-order mutants contain one syntactic change compared to the original program, i.e., one mutation operator is applied at one location in the source code.

Definition 4.11 (Higher-Order Mutant)
Higher-order mutants contain several mutations, i.e., mutation operators are applied several times.

DeMillo and Offutt [79] identified two conditions that must be fulfilled to kill a mutant:

Definition 4.12 (Necessity Condition)
The necessity condition says that the state of the mutated program after some execution of the mutated statement must be incorrect with respect to the original program. This implies that the mutated statement must be reached. This is necessary, but not sufficient.

Definition 4.13 (Sufficiency Condition)
The sufficiency condition says that the final state of the mutant must differ from the final state of the original program, i.e., the necessary incorrect intermediate state must propagate to an incorrect final state.

Both conditions have to hold in mutation testing as introduced by Hamlet [109] and DeMillo et al. [78]. It is often referred to as strong mutation testing:

Definition 4.14 (Strong Mutation Testing)
Strong mutation testing requires the necessity condition and the sufficiency condition to be fulfilled.

Later on, Howden [121] suggested weak mutation testing:

Definition 4.15 (Weak Mutation Testing)
In weak mutation testing, the satisfaction of the necessity condition alone is sufficient to kill a mutant.

Note that this distinction between strong and weak mutation testing is related to the concepts of faults (Definition 2.5), errors (Definition 2.6), and failures (Definition 2.7). In weak mutation testing, the identification of an error is enough to kill a mutant. The necessity condition correlates to reachability and infection (cf. Figure 2.1). In strong mutation testing, only identified failures cause a mutant to be killed. Here, the additionally required sufficiency condition corresponds to the propagation from an error to a failure. Note that weak mutation testing assumes access to the internals of the SUT. This is always the case in mutation testing. Otherwise no mutants could have been generated.

The advantage of weak mutation testing is that it requires less computational efforts as the program does not always have to be fully executed. On the other hand, it potentially results in less effective test sets. However, experimentation has shown that weak mutation testing can result in test suites that are almost as effective as test sets from strong mutation testing. Furthermore, it has been experienced that at least 50% of the execution time can be saved [23]. For example, one of these studies was conducted by Offutt and Lee [168]. They compared weak and strong mutation testing for unit testing. They came to the conclusion that weak mutation testing is a cost-effective alternative to strong mutation testing. However, for safety-critical applications they still recommend the use of strong mutation testing.

4.3 Model-Based Mutation Testing

In MBT, test cases are automatically derived from the test model. However, exhaustive testing, i.e., using all of the test cases that can possibly be created from a model, is impractical. Test selection criteria are required in order to select a proper subset of the possible test cases:

Definition 4.16 (Test (Selection) Criterion)
A test selection criterion, or shortly test criterion, provides information about what shall be tested. It is sometimes also referred to as test case specification.

Test selection criteria form one dimension of Utting et al.'s taxonomy of MBT (cf. Figure 4.3). Examples for test selection criteria are manifold. Often, some coverage criterion or random traversal on the test model are used. They do not involve extra effort, but do not systematically cover functionality. Some strategies use test purposes stating that for example a specific transition or a sequence of transitions shall be traversed. A disadvantage thereof is that they have to be designed manually.

We follow a fault-centred approach, i.e., use mutations for test case generation. This combination of MBT and mutation testing is called *model-based mutation testing*. Figure 4.5 depicts the process of model-based mutation testing, which is for the most part the same as in MBT (cf. Figure 4.1). The parts specific to model-based mutation testing are highlighted in yellow in Figure 4.5. The mutation concept is employed on the test model instead of the source code. Like in conventional mutation testing, the mutation can be fully automated via mutation operators that are defined for the model elements. Then, given the original model and a set of mutated models, we automatically generate test cases that kill the model mutants, i.e., reveal their non-conforming behaviour. This is accomplished by a conformance check between the original and the mutated models. The generated abstract test cases (cf. Definition 4.3) are then concretised and executed on the SUT and will detect whether one of the given faulty models has been implemented instead of the correct, original model. Hence, the generated test suite covers all of the failures caused by the model mutation operators and has a high chance of covering many additional similar failures. The underlying hypotheses are the same as those of classical mutation testing, i.e., the competent programmer hypothesis (cf. Definition 4.7) and the coupling effect (cf. Definition 4.8). Additionally, model-based mutation testing assumes that the mutated models sufficiently represent realistic faults in the SUT.

Example 4.1. Consider the CAS described in Section 1.6.1. Figure 4.6 depicts a possible mutation of the model. It replaces the exit action of the *Alarm* state. Originally, on exit of this state the alarms are

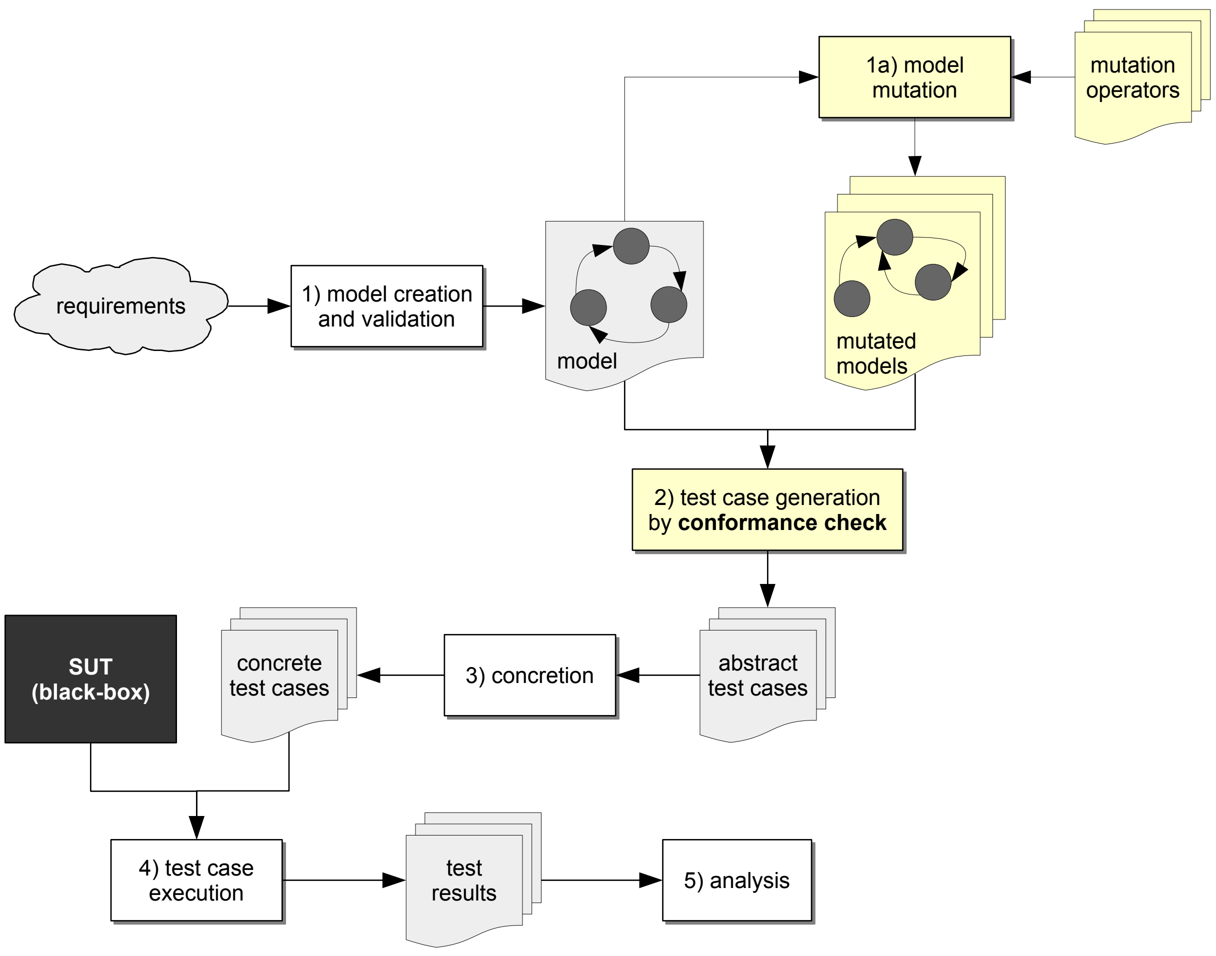


Figure 4.5: Model-based mutation testing: As in conventional MBT, test cases are derived from a formal model (cf. Figure 4.1). As a test specification, model-based mutation testing uses mutation operators to generate a set of mutated models. The test case generator performs a conformance check between the original and the mutated models to generate test cases that kill the model mutants, i.e., reveal their non-conforming behaviour.

deactivated. In our mutated version, the system will become armed. The left-hand side of Figure 4.7 shows a test case that distinguishes this mutated model from the original model. Note that we do not consider mutations of the testing interface (cf. Definition 2.9). However, this test case refers to a simpler testing interface than that introduced in Figure 1.3. Instead of individual actions for turning the sound and flash on/off, we consider one action *ActivateAlarms*, and one action *DeactivateAlarms*. Furthermore, arming and disarming the system is represented by *ShowArmed* and *ShowUnarmed* respectively. In this way, the actions directly correspond to the entry/exit actions in the state machine. In our test cases, controllable events are marked by prefix *ctr*, while observable events have the prefix *obs*. Furthermore, the first parameter of each event in our test cases denotes time. For controllable events, it states the number of time units the tester has to wait before sending the input to the SUT. For observable events, it denotes the number of time units after which the SUT might deliver an output.

The distinguishing test case in Figure 4.7 starts with closing and locking the doors. After 20 seconds, the system becomes armed. Afterwards, a door is opened causing the system to become disarmed and to activate the alarms (flash and sound) immediately. Then, the test case unlocks the car and the alarms are deactivated instantly. The test case is completed by a *pass* verdict after this observation. The right-hand side of Figure 4.7 shows the execution of this test case on the mutated model introduced in Figure 4.6.

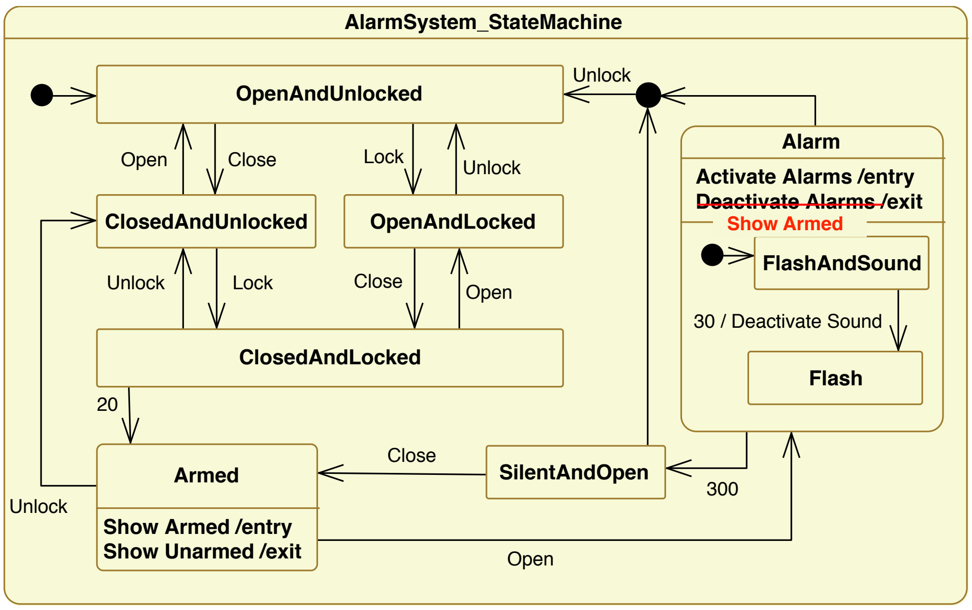


Figure 4.6: UML state machine of a mutated CAS: the original exit action of the *Alarm* state is *DeactivateAlarms*. It is replaced by the *ShowArmed* action.

Expected observable events are coloured in green. Unexpected observations are coloured in red indicating a fail verdict. Closing and locking all doors correctly causes the SUT to become armed after 20 seconds. By opening a door, the SUT becomes disarmed and activates the alarms, which is still correct. However, the next step of the test case is to unlock the car. This event triggers the incorrect behaviour of the model mutant. Instead of deactivating the alarms, the SUT gets armed and the test case fails. With our model-based mutation testing approach, we generate such test cases automatically. □

The used conformance relation is essential for model-based mutation testing in two ways. First of all, it specifies whether a SUT is correct with respect to its specification – in our case the (original) formal model of the SUT. This aspect is already known from conformance testing in general (cf. Chapter 3). The additional importance of the conformance relation in model-based mutation testing concerns test generation. The conformance relation defines if a syntactic change in a mutant represents a failure, i.e., if a mutated model conforms to the original model or not. Only if the mutated model does not conform to the original model, a distinguishing test case can be generated.

This implies that equivalent mutants (Definition 4.6) are a special case in model-based mutation testing – just like in conventional mutation testing. In our approach, equivalent model mutants are identified automatically. This is non-trivial as it involves an equivalence (conformance) check between original and mutated models. Since, equivalence is undecidable in general, we have to restrict ourselves to a bounded conformance check. This applies in two ways: (1) The domains of all variables in our models are bounded. (2) The behaviour of the original and the mutated models is compared up to a given bound.

Note that the aim of our work is not to test models, but to generate test cases that cover certain faults. Thereby, we work in a field that has not got very much attention yet. Jia and Harman describe the situation on mutation-based test case generation in their recent survey on mutation testing [130]: So far, much more effort has been spent on the definition of mutation operators and classical mutation testing and not so much work has been done on test case generation from mutations. Hence, *“at present, practical software test data generation for mutation test adequacy remains an unresolved problem”*. Furthermore, they identified a *“pressing need”* to address this problem. One particular issue is scalability, which is one topic of this thesis.

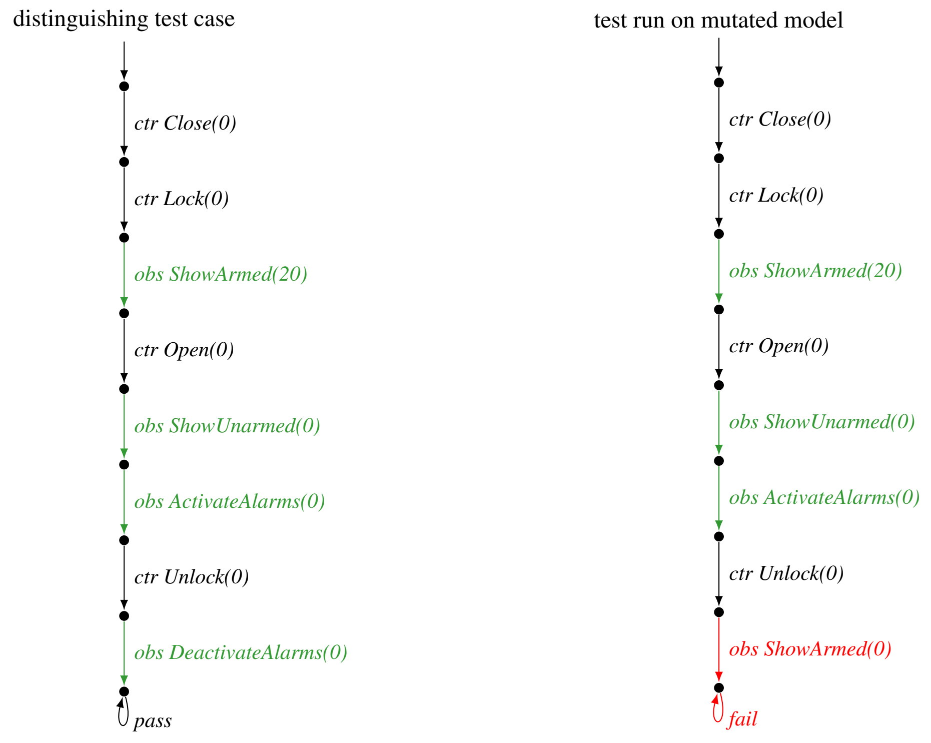


Figure 4.7: The test case on the left-hand side distinguishes the mutated CAS model (Figure 4.6) from the original model (Figure 1.4). The execution of this test case on the mutated model is shown at the right-hand side.

To conclude, model-based mutation testing is fault-centred. It rather aims at falsifying non-conformance than at verifying conformance. To cite Dijkstra [81]: *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."* Of course, it is true that testing cannot show the absence of all bugs. However, we can at least say which bugs are not present in a piece of software. Hence, model-based mutation testing can show the absence of *specific* bugs [6].

5 Action Systems

Parts of this chapter have been published in MBT 2012 [14], CSTVA 2012 [15], QSIC 2012 [13], and TAP 2013 [16].

This chapter gives an introduction to the original action system formalism. Subsequently, it presents our variant of action systems, which will be used throughout this work. Finally, it reviews extensions of action systems and other closely related formalisms.

5.1 Classical Action Systems

Action systems are a variant of Dijkstra's guarded command language [82] for modelling concurrent reactive systems. They have a formal semantics with refinement laws and are compositional. Many extensions exist, but the main idea is that a system state is updated by atomic, guarded actions that may be enabled or not. The action system continuously loops and determines which actions are enabled. If no action is enabled, the action system terminates. If several actions are enabled, one is chosen non-deterministically. Hence, concurrency is modelled in an interleaving semantics.

Action systems were introduced by Back and Kurki-Suonio in 1983 [28]. Since then, a lot of research has been conducted on action systems. For example, Back et al. defined a temporal logic framework [28, 29] and a refinement calculus framework [31, 33, 32] for action systems.

A common notation to represent an action system $\mathcal{A}$ is defined in the following.

Definition 5.1 (Classical Action System Notation)

$$\mathcal{A} = |[\ \textbf{var}\ v, w^*\ \bullet\ S_0\ ;\ \textbf{do}\ A_1\ [\,]\ ...\ [\,]\ A_n\ \textbf{od}\ ]|: I$$

The **var** section contains the variable declarations. The set of all declared variables defines the state space of the action system. Each variable is of a certain type. The set of variables v denotes variables that are internal to the action system. Variables marked with an asterisk w^* are exported by the action system and become global variables. They can be imported by other action systems via the import list I. S_0 is the initialisation action. The do-od block (**do** ... **od** construct) is the central part of the action system. It is basically a loop and contains the non-deterministic choice $[\,]$ over the actions $A_1, ..., A_n$. An action A_i is a guarded command [82]. It consists of a guard, which is a Boolean expression, and a body, which is composed of statements that will be shown later.

In order to facilitate the modelling of compositional systems, where each component can be modelled by an individual action system, the parallel composition of action systems has been defined.

Definition 5.2 (Parallel Composition of Action Systems)
For two action systems

$$\mathcal{A} = |[\ \textbf{var}\ v^{\mathcal{A}}, w^{\mathcal{A}*}\ \bullet\ S_0^{\mathcal{A}}\ ;\ \textbf{do}\ A_1^{\mathcal{A}}\ [\,]\ ...\ [\,]\ A_m^{\mathcal{A}}\ \textbf{od}\ ]|: I^{\mathcal{A}}$$

and

$$\mathcal{B} = |[\ \textbf{var}\ v^{\mathcal{B}}, w^{\mathcal{B}*}\ \bullet\ S_0^{\mathcal{B}}\ ;\ \textbf{do}\ A_1^{\mathcal{B}}\ [\,]\ ...\ [\,]\ A_n^{\mathcal{B}}\ \textbf{od}\ ]|: I^{\mathcal{B}}$$

with $v^{\mathcal{A}} \cap v^{\mathcal{B}} = \emptyset$, the parallel composition $\mathcal{C}$ is defined as follows:

$$\mathcal{C} = \mathcal{A}||\mathcal{B}\ =_{df}\ |[\ \textbf{var}\ v^{\mathcal{C}}, w^{\mathcal{C}*}\ \bullet\ S_0^{\mathcal{A}}; S_0^{\mathcal{B}}\ ;\ \textbf{do}\ A_1^{\mathcal{A}}\ [\,]\ ...\ [\,]\ A_m^{\mathcal{A}}\ \ [\,]\ \ A_1^{\mathcal{B}}\ [\,]\ ...\ [\,]\ A_n^{\mathcal{B}}\ \textbf{od}\ ]|: I^{\mathcal{C}}$$

with $v^{\mathcal{C}} = v^{\mathcal{A}} \cup v^{\mathcal{B}}$, $w^{\mathcal{C}*} = w^{\mathcal{A}*} \cup w^{\mathcal{B}*}$, and $I^{\mathcal{C}} = (I^{\mathcal{A}} \cup I^{\mathcal{B}}) \setminus (w^{\mathcal{A}*} \cup w^{\mathcal{B}*})$.

A pre-condition for parallel composition is that the sets of internal variables $v^{\mathcal{A}}$ and $v^{\mathcal{B}}$ are disjoint. For parallel composition, the new sets of internal and global variables $v^{\mathcal{C}}$ and $w^{\mathcal{C}*}$ are obtained by joining the respective individual variable sets. The initialisation is the sequential composition of the initialisations of $\mathcal{A}$ and $\mathcal{B}$. The new do-od block is the non-deterministic choice over all actions defined in both action systems. Finally, the set of imported variables is the union of the import sets of both action systems, excluding the union of those variables exported by the individual action systems.

Typically, the semantics of action systems is specified via weakest pre-conditions like originally defined by Dijkstra [82]. Pre- and post-conditions are predicates over the state variables of an action system. The weakest pre-condition predicate transformer $wp(S, Q)$ states the weakest pre-condition that must hold such that statement S can establish a given post-condition Q. Weakest pre-conditions are defined recursively. In the following, we give an overview of the most important action system statements and their weakest pre-conditions [31, 33].

Definition 5.3 (Weakest Pre-conditions)
Actions are defined as guarded commands $g \Rightarrow B$ where the guard g is a Boolean condition, and the body B is a sequential, possibly non-deterministic statement on the state variables of an action system. Furthermore, v denotes a state variable, and e is some expression over state variables. Assumptions and assertions are predicates over the state variables and denoted by a. Moreover, Q denotes a given post-condition over the state variables.

$$\begin{array}{lcll}
wp(g => B, Q) & =_{df} & g \Rightarrow wp(B, Q) & \text{(guarded command)} \\
wp(magic, Q) & =_{df} & true & \text{(miraculous action)} \\
wp(abort, Q) & =_{df} & false & \text{(aborting action)} \\
wp(skip, Q) & =_{df} & Q & \text{(stuttering action)} \\
wp(B_1; B_2, Q) & =_{df} & wp(B_1, wp(B_2, Q)) & \text{(sequential composition)} \\
wp(B_1 \; [] \; B_2, Q) & =_{df} & wp(B_1, Q) \wedge wp(B_2, Q) & \text{(non-deterministic choice)} \\
wp(v := e, Q) & =_{df} & Q[v \leftarrow e] & \text{(assignment)} \\
wp([a], Q) & =_{df} & a \Rightarrow Q & \text{(assumption)} \\
wp(\{a\}, Q) & =_{df} & a \wedge Q & \text{(assertion)}
\end{array}$$

The weakest pre-condition of an action, i.e., a guarded command $g => B$ and a post-condition Q is defined as implication (Equation *guarded command*). The action *magic* establishes any post-condition regardless of the pre-condition (Equation *miraculous action*). However, it is only a theoretical concept and not implementable. The action *abort* is used for modelling disallowed behaviour. Hence, its weakest pre-condition is false (Equation *aborting action*). The *skip* action does nothing (Equation *stuttering action*). Statements may be composed in sequence. The weakest pre-condition of a sequence of statements is calculated backwards: the weakest pre-condition of the second statement B_2 serves as the post-condition of the first statement B_1 (Equation *sequential composition*). For non-deterministic choices, the conjunction of both weakest pre-conditions has to hold, in order to guarantee post-condition Q regardless of which action has been chosen (Equation *non-deterministic choice*). For an assignment, the weakest pre-condition is obtained from Q by substituting the assigned variable v with expression e. This substitution is expressed by the notation $Q[v \leftarrow e]$ (Equation *assignment*). The weakest pre-condition of an assumption is defined by implication: the assumption implies the post-condition (Equation *assumption*). As a consequence, false assumptions lead to *magic* behaviour. The weakest pre-condition of an assertion is defined via conjunction (Equation *assertion*). Hence, false assertions lead to *abort*.

If an action is enabled, is determined by its enabledness guard [33]. Informally, it states that an action is not enabled in states where it leads to abort, i.e., post-condition false. Formally, it is defined as follows.

Definition 5.4 (Enabledness Guard)
The enabledness guard gA of an action A is defined as $gA = \neg wp(A, \mathit{false})$.

Skip, abort, and assignments are always enabled. Consider for example the skip action. Its enabledness guard is defined as $\neg wp(skip, \mathit{false})$. By using the definition of the weakest pre-condition of skip (Definition 5.3, Equation *stuttering action*), we have $\neg \mathit{false} \Leftrightarrow true$.

Additionally, the termination of an action is determined by its termination guard [33]. It ensures that some post-state is reached eventually. Formally, it is defined as follows.

Definition 5.5 (Termination Guard)
The termination guard tA of an action is defined as $tA = wp(A, true)$.

5.2 Action Systems in this Work

In the following, we describe the language for action systems that is underlying this work. Aside from a few minor changes, it has been established by colleagues during the MOGENTES project. The language was designed to facilitate the expression of UML state machine and OOAS constructs (cf. Section 5.4.1) in action systems and already served as input language for MoMuT::UML's existing test case generation backend Ulysses [10]. As Ulysses is implemented in Prolog, the concrete syntax has also been defined using Prolog language constructs. This accommodates parsing and further processing within Prolog. The language was given an operational semantics in terms of LTSs with inputs and outputs (cf. Definition 3.12). Both, the concrete syntax and the operational LTS semantics have only been described informally [140, 50, 10]. In this work, we formally present the syntax and give a relational predicative semantics for action systems, which is well-suited for our constraint-based approach. Our work was incremental. We started with a restricted language called *plain action systems* and later on extended this language by the missing features required for the integration into the MoMuT::UML tool chain. We refer to the latter as *complex action systems*.

We start this section with an example to illustrate our variant of action systems. Subsequently, we describe the concrete syntax and present our relational predicative semantics – first for our plain action systems, then for our complex action systems.

5.2.1 An Action System Modelling the Car Alarm System

Listing 5.1 shows code snippets from an action system modelling the CAS as described in Section 1.6.1. It is based on an OOAS originally created for the MOGENTES project by Willibald Krenn, who relied on the UML state machine depicted in Figure 1.4. Note that this particular action system is only one way to model the CAS. There are many other possibilities to specify the desired behaviour in an action system. For example, the state space could be reduced by a more efficient encoding. However, this model has been created in a real-world setting. It was developed incrementally, while the requirements still changed and hence it is not optimal. The full model can be found in Section A in the appendix.

After these general comments on the model, we explain its individual parts in the following. The first three lines of Listing 5.1 contain user-defined types, which are basically integers with restricted ranges. In Line 1, a type with name *enum_state* is defined. Its domain begins at 0 and ends at 7. Line 6 declares a variable with name *aState*, which is of type *enum_state*. It represents the states (including nested states) used in the UML state machine of the CAS (Figure 1.4). Additionally to integers, the Boolean data type *bool* with the elements true and false is supported (Line 4). In Line 8, three variables of type bool are declared. Note that lists are enclosed by square brackets as in Prolog. For example, $[\mathit{fromSilentAndOpen}, \mathit{flashOn}, \mathit{soundOn}]$ is a list with the three elements $\mathit{fromSilentAndOpen}$, $\mathit{flashOn}$, and $\mathit{soundOn}$. The *var* predicates in our Prolog syntax correspond to the *var* section in the

```
type(enum_state, X) :- X in 0..7.
type(int_0_4, X) :- X in 0..4.
type(int_0_270, X) :- X in 0..270.
type(bool, X) :- member(X, [true, false]).

var([aState], enum_state).
var([fromAlarm, fromArmed], int_0_4).
var([fromSilentAndOpen, flashOn, soundOn], bool).

state_def([aState, fromAlarm, fromArmed, fromSilentAndOpen, flashOn, soundOn]).

init([6, 0, 0, false, false, false]).

as :-
    actions (
    'ArmedOn'(Wait_time)::(true) =>
    (
       % case 1: arm the system after 20 seconds in closed and locked
        ((Wait_time #= 20 #/\ aState #= 3) => (aState := 2))
      []
       % case 2: arm the system immediately when closing the car after an alarm
        ((Wait_time #= 0 #/\ aState #= 2 #/\ fromSilentAndOpen #= true) =>
            (fromSilentAndOpen := false))
    ),
    'Lock'(Wait_time)::(true) =>
    (
       % case 1: in initial state, where car is open and unlocked
       ((aState #= 6 #/\ fromAlarm #= 0) => (aState := 5))
      []
       % case 2: in state, where car is closed and unlocked
       ((aState #= 4 #/\ fromArmed #\= 1) => (aState := 3; fromArmed := 0))
    ),
    ...
    ),
  dood (
      'Lock'(0)
  [] [T:int_0_270]:'ArmedOn'(T)
  [] ...
  ).
```

Listing 5.1: Code snippet from an action system model describing the CAS.

notation for classical action systems presented earlier (cf. Definition 5.1). However, we do not support composition (Definition 5.2) on the level of action systems. Hence, all variables are internal to the action system. These internal variables were denoted by v in the notation for classical action systems. Furthermore, there are neither exported variables, which were represented by the set w^*, nor imported variables, which were given by an import list I in Definition 5.1.

The reason why we do not need to address compositions of action systems is that composition is handled earlier in the MoMuT::UML tool chain (cf. Section 1.5.1). Action systems are only an intermediate format for the test case generation backends of MoMuT::UML. Originally, models are given as UML class diagrams and state machines. Of course, they may consist of several components. This compositional structure is also transferred to the intermediate Object-Oriented Action System (OOAS) model. However, during the transformation from the OOAS model into an action system, the parallel composition of the individual components is resolved and as a result, one single action system comprises the whole modelled functionality.

The structure of the state of the action system is defined in Line 10 in Listing 5.1. It is specified by a list of variables that have been declared earlier. In contrast to the classical notation, we do not automatically include all declared variables as part of the state space. This is sometimes helpful during the development of a model. Furthermore, the *init* predicate in Line 12 defines the initial values for the state. It corresponds to the initialisation action S_0 in the classical notation. At Line 14, the description of the behaviour of the action system begins. It consists of an *actions* block (Lines 15 to 34) and a *do-od* block (Lines 35 to 39). In the classical notation, the actions are directly given in the do-od block. In contrast, we decoupled the definitions of the actions in the *actions* block from their composition in the *do-od* block. This is possible since each action in our notation has a unique name, which can be used in the do-od block. The purpose of this separation will be seen later, when we extend our language and allow additional operators (not only non-deterministic choice) for the composition of actions in the do-od block.

Like in classical action systems, an action is a guarded command, but in our case it has a name and optionally also parameters. Hence, in our language an action consists of a name, optional parameters, a guard and a body: $name(parameter\ list) :: guard => body$ (cf. Lines 16 to 24 and Lines 25 to 32). The actions in our example have one parameter, which models the number of time units that have passed since the previous action. For example, the sequence $'Lock'(0)\ ;\ 'ArmedOn'(20)$ means that the car is being locked immediately after the previous action, and that $'ArmedOn'$ occurs 20 time units afterwards. It is also possible to have actions with more than one parameter or without any parameters. Sequential composition is represented by the ; operator. The operator [] denotes non-deterministic choice. We use it in our example together with guarded commands to distinguish between different cases in which an action is enabled. Consider for example the definition of the action $'ArmedOn'$ (Lines 16 to 24). We distinguish between two situations. In the first case (Lines 18 and 19), $'ArmedOn'$ has to be fired after 20 seconds in the state where the car is closed and locked. The guard constrains both the time parameter ($Wait\_time\ \#= 20$) as well as the condition that the state of the system must be closed and locked ($aState\ \#= 3$). The body of the guarded command updates the state variables of the action system. In this case, we move on to a state that represents that the system is armed ($aState := 2$). The other case is modelled analogously. It refers to the situation in which the alarms were triggered by opening the armed car without unlocking it before. After a timeout, the alarms are deactivated and the system becomes armed immediately after closing the doors. This last part is modelled in the Lines 22 to 23.

The do-od block (Lines 35 to 39) connects previously defined actions via non-deterministic choice. Note that the action $'Lock'$ may only occur without waiting. This is accomplished by using the constant value 0 as parameter for $'Lock'$ in the do-od block. Thus, it is not necessary to further restrict the $Wait\_time$ parameter in the guards of the definition of $'Lock'$ (Lines 28 and 31). The action $'ArmedOn'$ has a variable T as parameter. It is restricted to be an integer in the range of 0 to 270. The parameter is further constrained to be either 20 or 0 in the guards of the definition of $'ArmedOn'$ (Lines 19 and 22). The execution of an action system is a continuous iteration over the do-od block. In each iteration, an enabled action is non-deterministically chosen for execution. In our example, at least one action is always enabled. Hence, the CAS never terminates, but continuously interacts with its environment.

In the following, we present the syntax and semantics of a first subset of our action system language, which we refer to as plain action systems.

5.2.2 Plain Action Systems

Syntax. The above example already gave an impression of our syntax for action systems. Figure 5.1 gives a formal definition of the syntax for our *plain action systems*, which is restricted and has been used as a starting point for our work. An action system AS consists of basic definitions D, action definitions $\overline{A}$, and a do-od block P. Note that the overbar notation is used to represent sequences of the given elements. Hence, $\overline{A}$ denotes a sequence of action definitions, where one action definition is given by A.

$$
\begin{aligned}
AS &::= D\ \mathtt{\textbf{as}} :\!-\ \mathtt{\textbf{actions}}(\overline{A}),\ \mathtt{\textbf{dood}}(P).\\
D &::= \overline{\mathtt{\textbf{type}}(t, X) :\!-\ X\ \mathtt{\textbf{in}}\ n_1..n_2.}\ \ \overline{\mathtt{\textbf{var}}([\overline{v}], t).}\ \ \mathtt{\textbf{state\_def}}([\overline{v}]).\ \mathtt{\textbf{init}}([\overline{c}]).\ \mathtt{\textbf{input}}([\overline{l}]).\ \mathtt{\textbf{output}}([\overline{l}]).\\
A &::= L :: g => B\\
L &::= l \mid l(\overline{X})\\
g &::= e \mid e \vee e \mid e \wedge e \mid e = e \mid ...\\
B &::= v := e \mid g => B \mid skip \mid B; B \mid B\,[\,]\,B\\
e &::= v \mid c \mid X \mid e + e \mid ...\\
P &::= E \mid E\,[\,]\,P\\
E &::= l \mid [\overline{X : t}] : l(\overline{c \,|\, X})\\
X &::= Prolog\ variable
\end{aligned}
$$

Figure 5.1: Syntax for plain action systems.

The basic definitions D comprise the definition of types t, the declaration of variables v of type t, the definition of the system state as a variable vector $\overline{v}$, and the definition of the initial state as a vector of constants $\overline{c}$. Furthermore, it states which actions represent inputs for the SUT and which actions represent outputs of the SUT to define the testing interface (cf. Definition 2.9). Both input as well as output actions are stated at the end of the definitions D as lists of action labels l ($\mathtt{\textbf{input}}([\overline{l}]).\ \mathtt{\textbf{output}}([\overline{l}]).$), which must not overlap.

An action A is a labelled guarded command with label L, guard g and body B. Actions may have a list of parameters $\overline{X}$. An action's guard is basically a condition over constants, state variables, and parameters. We support common operators like disjunction, conjunction, equality, etc., as well as arithmetic operators such as addition, subtraction, etc. The body of an action may assign an expression e to a variable v, it may do nothing ($skip$), or it may comprise guarded commands. Action bodies may be composed by sequential composition ; or non-deterministic choice $[\,]$. The do-od block P provides an event-based view on the action system. It composes the actions by their action labels l and optional parameters via non-deterministic choice. If necessary, it also defines the data types of local variables used as parameters for actions.

Although our concrete syntax is rather different to the notation for classical action systems given in Definition 5.1, both languages are similar in the following aspects. Action systems have variables that constitute its state space, an initialisation construct, and actions, which are guarded commands that are non-deterministically composed in a do-od block. The greatest difference between the classical notation and our language is the labelling of the actions and optional parametrisation. However, assigning names to actions is not new. For example, labels for actions have already been used to link action systems and the process algebra Communicating Sequential Processes (CSP) [159, 61].

Semantics. For constraint-based approaches [103, 70, 208, 173], the semantics of programs is often encoded via Static Single Assignment (SSA) form [20]. Yet for our refinement check, the SSA form is not suitable as will be demonstrated in Section 6.4.2. For action systems, the formal semantics is typically defined in terms of weakest pre-conditions (cf. Definition 5.3). However, for our constraint-based approach we chose a relational predicative semantics, which is very similar to a constraint satisfaction problem. We follow the style of He and Hoare's Unifying Theories of Programming (UTP) [119], where predicates are used to describe the relations between observations before and after execution of a pro-

$$\begin{array}{lll}
\phi(l :: g => B) & =_{df} & g \ \wedge\ \phi(B)\ \wedge\ tr' = tr^\frown[l] \\
\phi(l(\overline{X}) :: g => B) & =_{df} & \exists\, \overline{X} : (g\ \wedge\ \phi(B)\ \wedge\ tr' = tr^\frown[l(\overline{X})]) \\
\phi(g => B) & =_{df} & g\ \wedge\ \phi(B) \\
\phi(v_1 := e) & =_{df} & v_1' = e\ \wedge\ v_2' = v_2\ \wedge \cdots \wedge\ v_n' = v_n \\
 & & \text{with } \overline{v} = \langle v_1, v_2, \ldots, v_n\rangle \text{ and } \overline{v}' = \langle v_1', v_2', \ldots, v_n'\rangle \\
\phi(B_1; B_2) & =_{df} & \exists\, \overline{v_0} : \phi(B_1)[\overline{v}' \leftarrow \overline{v_0}]\ \wedge\ \phi(B_2)[\overline{v} \leftarrow \overline{v_0}] \\
\phi(B_1\ [\,]\ B_2) & =_{df} & \phi(B_1)\ \vee\ \phi(B_2) \\
\phi(skip) & =_{df} & \overline{v}' = \overline{v} \\
\phi(B_1\ //\ B_2) & =_{df} & \phi(B_1)\ \vee\ (\neg\, \phi(B_1)\ \wedge\ \phi(B_2)) \\
\phi(\backslash m) & =_{df} & \phi(getB(m)) \\
\phi(obj\backslash m) & =_{df} & \phi(getB(obj\_m)) \\
\phi(v\backslash m) & =_{df} & \bigvee_{obj \in getClass(v)} (v = obj \wedge \phi(getB(obj\_m)))
\end{array}$$

Figure 5.2: Predicative semantics of actions: lines written in black are already relevant to plain action systems (Figure 5.1), lines written in blue become only relevant for complex action systems (Figure 5.3).

gram. A predicate p over a set of observations before execution $\overline{v} = \langle x, y, \ldots\rangle$ and a set of observations after execution $\overline{v}' = \langle x', y', \ldots\rangle$ is represented as $p(\overline{v}, \overline{v}')$. Note that all variables in $\overline{v}$ and $\overline{v}'$ denote relevant observations and occur as free variables in p. They are called the alphabet of predicate p.

Figure 5.2 presents our formal semantics of our actions in the form of a function ϕ, which maps concrete syntax to predicates. These predicates relate the pre-state of variables $\overline{v}$ and their post-state $\overline{v}'$ and represent the state changes of actions. Furthermore, the labels form a visible trace of events tr that is updated to tr' whenever an action runs through. Hence, the alphabet of predicates representing labelled actions comprises both the state variables before and after execution ($\overline{v}$ and $\overline{v}'$) as well as the trace of performed actions before and after execution (tr and tr'). A guarded action's transition relation is defined as the conjunction of its guard g, its body B, and the consecution of the action's label l to the previously observed trace. Note that the guard g is a condition and directly represents a predicate. In case of parameters $\overline{X}$, these are added as local variables to the predicate. An assignment updates one variable v_1 with the value of an expression e and leaves the rest unchanged. For sequential composition, there must exist an intermediate state $\overline{v_0}$ that can be reached from the first body predicate $\phi(B_1)$ and from which the second body predicate $\phi(B_2)$ can lead to its final state. Note that our notation uses substitutions to represent this. The post-state $\overline{v}'$ of the first body predicate $\phi(B_1)$ is substituted by $\overline{v_0}$. Furthermore, the pre-state of the second body predicate $\phi(B_2)$ is also substituted by $\overline{v_0}$. Hence, the alphabet of $\phi(B_1)$ comprises $\overline{v}$ and $\overline{v_0}$, and the alphabet of $\phi(B_2)$ comprises $\overline{v_0}$ and $\overline{v}'$. The alphabet of the overall sequential composition is $\overline{v}$ and $\overline{v}'$ again. Non-deterministic choice is defined as disjunction. Finally, the $skip$ statement does not change anything, i.e., $\overline{v}' = \overline{v}$. The just described constructs are relevant for our plain action system language and are written in black in Figure 5.2. Blue parts only concern complex action systems and will be described in the next section.

The semantics of the do-od block of plain action systems is as follows: while actions are enabled in the current state, one of the enabled actions is non-deterministically chosen and executed. An action is enabled in a state if it can run through, i.e., if a post-state exists such that its semantic predicate can be satisfied. The action system terminates if no action is enabled.

$$
\begin{aligned}
AS &::= D\ \texttt{as} :\!-\ \texttt{methods}(\overline{M}),\ \texttt{actions}(\overline{A}),\ \texttt{dood}(P).\\
D &::= \overline{T}\ \overline{\texttt{var}([\overline{v}], t).}\ \texttt{state\_def}([\overline{v}]).\ \texttt{init}([\overline{c}]).\ \texttt{input}([\overline{l}]).\ \texttt{output}([\overline{l}]).\\
T &::= \texttt{type}(t, X) :\!-\ X\ \texttt{in}\ n_1..n_2.\ \mid\ \texttt{type}(t, X) :\!-\ \mathit{member}(X, [\overline{obj}]).\\
M &::= H = B_m\\
H &::= m \mid m(\overline{X}) \mid \mathit{obj\_m} \mid \mathit{obj\_m}(\overline{X})\\
A &::= L :: g => B\\
L &::= l \mid l(\overline{X})\\
g &::= e \mid e \vee e \mid e \wedge e \mid e = e \mid \ldots \mid C \mid C(\overline{X})\\
B &::= v := e \mid g => B \mid \mathit{skip} \mid B\ \mathit{Op}\ B \mid C \mid C(\overline{X})\\
e &::= v \mid c \mid X \mid e + e \mid \ldots\\
B_m &::= B \mid \mathit{unify}(X, e)\\
C &::= \backslash m \mid \mathit{obj}\backslash m \mid v\backslash m\\
P &::= l \mid l(\overline{c}) \mid i(l) \mid i(l(\overline{c})) \mid [\overline{X : t}] : P_1 \mid P\ \mathit{Op}\ P\\
P_1 &::= E \mid i(E) \mid P_1\ \mathit{Op}\ P_1\\
E &::= l \mid l(\overline{c \mid X})\\
\mathit{Op} &::= [\,] \mid ; \mid //\\
X &::= \mathit{Prolog\ variable}
\end{aligned}
$$

Figure 5.3: Syntax for complex action systems: extensions with respect to plain action systems (Figure 5.1) are highlighted in blue.

5.2.3 Complex Action Systems

We had to extend our plain action systems to a richer language, which is required for interoperability with the already existing MoMuT::UML tool chain presented in Section 1.5.1. In MoMuT::UML, action systems are derived from UML state machines via Object-Oriented Action Systems (OOASs), which are briefly presented later on (Section 5.4.1) in the context of action system extensions and related formalisms. Aside from a few minor changes, the extended language has already been defined prior to this work with the intent to facilitate the transformation from UML state machines via OOASs into action systems. We refer to this extended language as *complex action systems* and describe its syntax and semantics in the following.

Syntax. The syntax for complex action systems is presented in Figure 5.3. Modifications with respect to our plain action systems (cf. Figure 5.1) are highlighted in blue. The first extension affects the data type definitions $\overline{T}$. In plain action systems, a data type T could only be an integer with a defined range, cf. first option for T in Figure 5.3: $\texttt{type}(t, X) :\!-\ X\ \texttt{in}\ n_1..n_2$). For the representation of the classes of an OOAS, a special enumeration data type has been introduced as a second option for T: $\texttt{type}(t, X) :\!-\ \mathit{member}(X, [\overline{obj}])$. In this way, the possible instances of a class can be represented as members of lists containing object identifiers (obj). Note that OOASs used in MoMuT::UML do not allow dynamic object creation or destruction (cf. Section 5.4.1). Hence, all possible objects are known in advance and can be enumerated as a finite list of object identifiers. The member variables of an object are given as state variables of the action system and are prefixed by the object identifier.

Another extension is the introduction of methods. Our action systems are not object-oriented. However, in MoMuT::UML they are derived from object-oriented models (UML state machines and OOASs). The translation of OOASs into our complex action systems creates one method per object, i.e., each possible object of a class has its own definitions of the methods of its class. Each method is prefixed by the object identifier and directly refers to the object's member variables, which are state variables of the action system that are also prefixed with the object's identifier. This is possible since there is no dynamic creation or destruction of objects in MoMuT::UML's OOASs (cf. Section 5.4.1). Hence, *methods* in our complex action systems are basically procedures. The concept of procedures in action systems is no new invention. Already in the 1990s, procedures have been introduced to action systems as a means of communication between action systems. As we do not consider compositions of action systems, we do not use procedures for the purpose of communication, but for encapsulation. The method definitions $\overline{M}$ have been inserted into the action system definition AS before the definition of the actions. Each method M consists of a head H and a body B_m. Each method's head H consists of a name m possibly followed by parameters $\overline{X}$. When derived from an OOAS, the method name has a prefix obj that refers to an object identifier. A method's body B_m is basically the same as an action's body B, i.e., it may contain assignments, guarded commands, etc. However, it furthermore allows for the unification of parameters with expressions, which is used to express return values. Methods may be called in the body of an action or method. For the use in guards, methods must be side-effect free, i.e., they must not contain assignments to state variables. For the sake of simplicity, we did not explicitly express this in our overview of the syntax. As already mentioned, methods may be called in method/action bodies or in guards, and may have parameters. For a method call C, the $\backslash$ operator is used. The expression $\backslash m$ denotes the call of a procedure with name m. When derived from OOASs, methods have a prefix for the object they are called with. This is either an object identifier *obj* like in $obj\backslash m$ or a state variable v containing an object like in $v\backslash m$.

Furthermore, the body B of actions or methods has been enriched by an additional operator $//$ for prioritising compositions, which was formally presented by Sekerinski and Sere [181]. A prioritising composition $B_1 \ // \ B_2$ gives priority to the left-hand side B_1. The right-hand side B_2 is only chosen if the left-hand side B_1 is not enabled. Hence, in contrast to non-deterministic choice, prioritising composition is deterministic.

Another major difference to plain action systems affects the do-od block P, which has been completely restructured. Previously, the do-od block composed actions solely via the non-deterministic choice operator and possibly contained definitions of local variables that serve as arguments for the actions. In complex action systems, the do-od block may also use sequential composition and the newly introduced prioritising composition to compose actions. Furthermore, the definitions of local variables may range over compositions of actions, not only over one action as before. However, note that parameters for actions must either be constants or must be defined as local variables (therefore the additional rule P_1). This enhanced structure of the do-od block is the main reason why we decoupled the definitions of the individual actions from the do-od block. Actions are defined only once and can be composed in the do-od block in a flexible manner.

Our last extension concerns internal actions. In complex action systems, an action label l in the do-od block may be enclosed in $i(l)$, which expresses that this action is internal and hidden. Internal events are a well-known concept in modelling. They correspond to τ-transitions in LTS [137]. In Communicating Sequential Processes (CSP) [118, 175], there also exists a hiding operator to hide a set of events. Butler introduced internal actions to action systems [61].

Semantics. For our complex action systems, add-ons for the semantics of the actions have been included in blue font in Figure 5.2. Prioritising composition $B_1 \ // \ B_2$ expresses prioritisation of B_1 over

B_2. Whenever B_1 is enabled, it will be chosen. If B_1 is not enabled, B_2 is executed provided that it is enabled. Just like for non-deterministic choices, if neither B_1 nor B_2 is enabled, the whole prioritising composition is not enabled. This is expressed by $\phi(B_1) \vee (\neg\phi(B_1) \wedge \phi(B_2))$.

Method calls are inlined. So basically, the semantics of a method call consists of its body's semantics. We assume a function $getB$ that returns the body for a given method name. For plain procedure calls $\backslash m$, finding the definition including the body is straightforward as it is identified by the name m. For method calls on object identifiers $obj\backslash m$, the according method definition is identified by obj_m, i.e., the backslash from the call is replaced by an underline. For methods called on state variables $v\backslash m$, we have to find the data type of the variable and have to check whether it is representing a class. In general, we cannot decide statically which concrete object is assigned to the used state variable. This may only be known at execution time. Hence, we have to consider each possible object and construct a disjunction over all possibly called methods. Each method call is guarded by a constraint that checks at runtime which object, i.e., which method call has to be activated ($v = obj$). We assume a function $getClass$, which takes a state variable, checks whether its type represents a class, and returns a set of all object identifiers associated with this class. This is inefficient as nesting might cause an exponential blowup of possibilities. However, this design of methods is predefined by MoMuT::UML. Figure 5.2 does not include the full semantics for method parameters, which was also specified by MoMuT::UML. Parameters for methods are passed by Prolog's unification. Actual parameters are either constants or local variables represented as Prolog variables. Furthermore, formal parameters are also represented as Prolog variables. Hence, we can unify formal and actual parameters. Note that actual parameters that are not constants, but Prolog variables may be used as return values by unifying them with a value or expression in the method body.

Internal actions are almost treated like normal actions. To be enabled, their guard and their body must hold. However, the action's label and possibly existing parameters are not added to the trace variable tr, but the special label τ. This τ-action is always of arity zero, because parameters of internal actions are also hidden.

The extended do-od block may contain non-deterministic choices as well as prioritising and sequential compositions of actions. Like for conventional action systems, the do-od block determines enabledness and loops until nothing is enabled any more. We changed the type of the constructs for which enabledness has to be decided. Previously, these were plain actions. Now, actions and nested constructs of sequential compositions, prioritising compositions, and non-deterministic choices of actions have to be considered. The enabledness thereof is defined via the semantics of the used actions and operators.

The extension of the do-od block by sequential and prioritising composition facilitates the translation of UML state machine constructs. For example, exit actions are executed on exit of a state. They can be easily expressed by the sequential composition of the exit action with the outgoing transition, which is also mapped to an action. Prioritising compositions mainly help to keep action guards simpler. Consider the following example. Let v and $exec$ be Boolean variables. The action $reset$ is only enabled if the $exec$ flag is true: $reset :: (exec) \Rightarrow (v := false)$. Furthermore, another very simple action toggles the value of the Boolean variable v. It is defined as $toggle :: (true) \Rightarrow (v := \neg v)$. It shall only be enabled if the $reset$ action is not enabled. This can be achieved by using prioritising composition in the do-od block: $reset\ //\ toggle$. If only non-deterministic choice was available ($reset\ []\ toggle$), then the guard of the $toggle$ action would need to be strengthened to $exec = false$. Note that the $exec$ flag may be set by additional actions in the action system. For translating UML state machines to OOASss and action systems respectively, prioritising composition is often used in the context of concurrency, e.g., for broadcasting events over several orthogonal regions. For further explanations on the transformation of UML state machines to OOASs and action systems, we refer to Krenn et al. [140].

5.3 Relating Predicates and Weakest Pre-Conditions

In the following, we discuss the relation between our predicative semantics (Figure 5.2) and the original weakest pre-condition semantics (Definition 5.3) for action systems. In their book on the Unifying Theories of Programming (UTP) [119], Hoare and He defined the weakest pre-condition of a statement B for a given post-condition Q by predicates:

$$wp(B,Q) \;\; =_{df} \;\; \neg(B;\neg Q)$$

We use this relation between weakest pre-conditions and predicates to show the equivalence of our predicative semantics and the weakest pre-condition semantics for common statements in the languages (skip statements, assignments, guarded commands, sequential compositions, and non-deterministic choices).

Theorem 5.1 (Equivalence of Semantics)
Our predicative semantics of action systems is equivalent to the weakest pre-condition semantics of action systems in the sense that

$$wp(B,Q) = \neg\phi(B \; ; \; \neg Q)$$

for $B \in \{skip,\;\; v := e,\;\; g => B_1,\;\; B_1 \,;\, B_2,\;\; B_1 \,[\,]\, B_2\}$ *with* $\phi(Q) = Q$.

Proof: We prove Theorem 5.1 by induction over the recursive definition of B, with $skip$ and $v := e$ forming the base cases. For the recursively defined cases $g => B_1,\; B_1 \,;\, B_2$ and $B_1 \,[\,]\, B_2$, we can then rely on $wp(B_1, Q_1) = \neg\phi(B_1 \; ; \; \neg Q_1)$ and $wp(B_2, Q_2) = \neg\phi(B_2 \; ; \; \neg Q_2)$ as induction hypotheses.

Stuttering action $skip$:

$\quad \neg\phi(skip \; ; \; \neg Q)$ {Fig. 5.2: $\phi(B_1 \,;\, B_2)$}

$= \neg\Big(\exists\, \overline{v_0} : \big(\phi(skip)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {Fig. 5.2: $\phi(skip)$, $\phi(Q)$}

$= \neg\Big(\exists\, \overline{v_0} : \big((\overline{v}' = \overline{v})[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \neg Q[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {apply $[\overline{v}' \leftarrow \overline{v_0}]$}

$= \neg\Big(\exists\, \overline{v_0} : \big((\overline{v_0} = \overline{v}) \;\wedge\; \neg Q[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {apply $\overline{v_0} = \overline{v}$}

$= \neg\Big(\exists\, \overline{v_0} : \big((\overline{v_0} = \overline{v}) \;\wedge\; \neg Q[\overline{v} \leftarrow \overline{v}]\big)\Big)$ {Q does not depend on $\overline{v_0}$ }

$= \neg\big(\neg Q \wedge \exists\, \overline{v_0} : (\overline{v_0} = \overline{v})\big)$ {simplify}

$= \neg(\neg Q \wedge true) \quad = \quad Q$ {Def. 5.3: stuttering action}

$= wp(skip, Q)$

Assignment $v_1 := e$:

$\quad \neg\phi((v_1 := e) \; ; \; \neg Q)$ {Fig. 5.2: $\phi(B_1 \,;\, B_2)$}

$= \neg\Big(\exists\, \overline{v_0} : \big(\phi(v_1 := e)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {Fig. 5.2: $\phi(v_1 := e)$, $\phi(Q)$}

$= \neg\Big(\exists\, \overline{v_0} : \big((v_1' = e \;\wedge v_2' = v_2 \wedge \cdots \wedge\; v_n' = v_n)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \neg Q[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {apply $[\overline{v}' \leftarrow \overline{v_0}]$}

$= \neg\Big(\exists\, \overline{v_0} : \big((v_{0,1} = e \wedge v_{0,2} = v_2 \wedge \cdots \wedge v_{0,n} = v_n) \wedge \neg Q[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {apply $\overline{v_0} = (e, v_2, \ldots, v_n)$}

$= \neg\Big(\exists\, \overline{v_0} : \big((v_{0,1} = e \wedge v_{0,2} = v_2 \wedge \cdots \wedge v_{0,n} = v_n) \wedge \neg Q[\overline{v} \leftarrow (e, v_2, \ldots, v_n)]\big)\Big)$ {move $\neg Q$}

$= \neg\big(\neg Q[\overline{v} \leftarrow (e, v_2, \ldots, v_n)] \wedge \exists\, \overline{v_0} : (v_{0,1} = e \;\wedge\; v_{0,2} = v_2 \;\wedge \cdots \wedge\; v_{0,n} = v_n)\big)$ {simplify}

$= \neg\big(\neg Q[\overline{v} \leftarrow (e, v_2 \ldots, v_n)] \wedge \mathit{true}\big) \quad = \quad Q[v_1 \leftarrow e]$ {Def. 5.3: assignment}

$= wp(v_1 := e, Q)$

Guarded command $g => B$:

$\neg\phi((g => B)\ ;\ \neg Q)$ {Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\Big(\exists\,\overline{v_0} : \big(\phi(g => B)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {Fig. 5.2: $\phi(g => B)$}

$= \neg\Big(\exists\,\overline{v_0} : \Big(\big(g\ \wedge\ \phi(B)\big)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big)\Big)$ {g does not depend on $\overline{v_0}$}

$= \neg\Big(g\ \wedge \exists\,\overline{v_0} : \big(\phi(B)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\big(g\ \wedge \phi(B\ ;\ \neg Q)\big)$ {induction hypothesis}

$= \neg\big(g\ \wedge \neg wp(B, Q)\big) \quad = \quad g \Rightarrow wp(B, Q)$ {Def. 5.3: guarded command}

$= wp(g => B, Q)$

Sequential composition $B_1 ; B_2$:

$\neg\phi\big((B_1\ ;\ B_2)\ ;\ \neg Q\big)$ {Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\Big(\exists\,\overline{v_0} : \big(\phi(B_1\ ;\ B_2)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\Big[\exists\,\overline{v_0} : \Big[\Big(\exists\,\overline{v_1} : \big(\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \phi(B_2)[\overline{v} \leftarrow \overline{v_1}]\big)\Big)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big]\Big]$ {do subst.}

$= \neg\Big[\exists\,\overline{v_0} : \Big[\Big(\exists\,\overline{v_1} : \big(\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \phi(B_2)[\overline{v} \leftarrow \overline{v_1}][\overline{v}' \leftarrow \overline{v_0}]\big)\Big) \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big]\Big]$ {widen $\exists\overline{v_1}$}

$= \neg\Big[\exists\,\overline{v_0} : \Big[\exists\,\overline{v_1} : \Big(\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \phi(B_2)[\overline{v} \leftarrow \overline{v_1}][\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big)\Big]\Big]$ {swap $\exists$s}

$= \neg\Big[\exists\,\overline{v_1} : \Big[\exists\,\overline{v_0} : \Big(\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \phi(B_2)[\overline{v} \leftarrow \overline{v_1}][\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big)\Big]\Big]$ {move B_1}

$= \neg\Big[\exists\,\overline{v_1} : \Big[\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \exists\,\overline{v_0} : \Big(\phi(B_2)[\overline{v} \leftarrow \overline{v_1}][\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big)\Big]\Big]$ {mod. subst.}

$= \neg\Big[\exists\,\overline{v_1} : \Big[\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \Big(\exists\,\overline{v_0} : \big(\phi(B_2)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)[\overline{v} \leftarrow \overline{v_1}]\Big]\Big]$

{Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\Big[\exists\,\overline{v_1} : \Big[\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \phi(B_2\ ;\ \neg Q)[\overline{v} \leftarrow \overline{v_1}]\Big]\Big]$ {induction hypothesis}

$= \neg\Big[\exists\,\overline{v_1} : \Big[\phi(B_1)[\overline{v}' \leftarrow \overline{v_1}] \wedge \big(\neg wp(B_2, Q)\big)[\overline{v} \leftarrow \overline{v_1}]\Big]\Big]$ {Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\phi\big(B_1\ ;\ \neg wp(B_2, Q)\big)$ {induction hypothesis}

$= wp\big(B_1, wp(B_2, Q)\big)$ {Def. 5.3: sequential composition}

$= wp(B_1\ ;\ B_2, Q)$

Non-deterministic choice $B_1\ [\,]\ B_2$:

$\neg\phi\big((B_1\ [\,]\ B_2)\ ;\ \neg Q\big)$ {Fig. 5.2: $\phi(B_1 ; B_2)$}

$= \neg\Big(\exists\,\overline{v_0} : \big(\phi(B_1\ [\,]\ B_2)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)$ {Fig. 5.2: $\phi(B_1\ [\,]\ B_2)$}

$$= \neg\Big(\exists\, \overline{v_0} : \Big(\big(\phi(B_1) \vee \phi(B_2)\big)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\Big)\Big) \qquad \{\text{distribute } \phi(\neg Q)\}$$

$$= \neg\Big(\exists\, \overline{v_0} : \Big(\big(\phi(B_1)[\overline{v}' \leftarrow \overline{v_0}] \wedge \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big) \vee \big(\phi(B_2)[\overline{v}' \leftarrow \overline{v_0}] \wedge \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)\Big) \quad \{\text{split } \exists\}$$

$$= \neg\Big(\Big(\exists\, \overline{v_0} : \big(\phi(B_1)[\overline{v}' \leftarrow \overline{v_0}] \wedge \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big) \vee \Big(\exists\, \overline{v_0} : \big(\phi(B_2)[\overline{v}' \leftarrow \overline{v_0}] \wedge \phi(\neg Q)[\overline{v} \leftarrow \overline{v_0}]\big)\Big)\Big)$$

$$\{\text{apply Fig. 5.2: } \phi(B_1\ ;\ B_2) \text{ two times}\}$$

$$= \neg\big(\phi(B_1\ ;\ \neg Q) \vee \phi(B_2\ ;\ \neg Q)\big) \qquad \{\text{induction hypothesis, two times}\}$$

$$= \neg\big(\neg wp(B_1, Q) \vee \neg wp(B_2, Q)\big) \quad = \quad wp(B_1, Q) \wedge wp(B_2, Q) \qquad \square$$

The following corollary shows the relation between the enabledness guard in terms of weakest pre-conditions (Definition 5.4) and the satisfiability of our semantic predicates.

Corollary 5.1 (Equivalence of Enabledness)
The enabledness in our predicative semantics of action systems is equivalent to the enabledness in the weakest pre-condition semantics in the sense that

$$\neg wp(B, \mathit{false}) = \exists\, \overline{v_0} : \phi(B)[\overline{v}' \leftarrow \overline{v_0}]$$

for $B \in \{\mathit{skip},\ \ v := e,\ \ g \Rightarrow B_1,\ \ B_1\,;\,B_2,\ \ B_1\,[\,]\,B_2\}$*. Existentially quantifying the pre-state* $\overline{v}$ *in the right-hand side of the equation, i.e.,* $\exists\, \overline{v}, \overline{v_0} : \phi(B)[\overline{v}' \leftarrow \overline{v_0}]$*, corresponds to a satisfiability check of* $\phi(B)$*.*

Proof:

$$\neg wp(B, \mathit{false}) \qquad \{\text{Theorem 5.1}\}$$

$$= \neg\neg\phi(B\ ;\ \neg \mathit{false}) \qquad \{\text{simplify}\}$$

$$= \phi(B\ ;\ \mathit{true}) \qquad \{\text{Fig. 5.2: } \phi(B_1\ ;\ B_2)\}$$

$$= \exists\, \overline{v_0} : \big(\phi(B)[\overline{v}' \leftarrow \overline{v_0}] \ \wedge\ \phi(\mathit{true})[\overline{v} \leftarrow \overline{v_0}]\big) \qquad \{\text{simplify}\}$$

$$= \exists\, \overline{v_0} : \big(\phi(B)[\overline{v}' \leftarrow \overline{v_0}]\big) \qquad \square$$

Regarding termination, we assume that the step relation, i.e., one iteration of the do-od block always terminates. This is a valid assumption as our language does not include abort statements or assertions. Therefore, it is sufficient to use relations instead of UTP's designs [119]. Relations do not explicitly consider termination, while designs have special observations *ok* and *ok′* indicating that the program has started and terminated respectively. In the following, we show that the termination guard (Definition 5.5) for our predicates is always true.

Corollary 5.2 (Termination Guard True)
The termination guard of an action B *in our predicative semantics of action systems is always* true*, i.e.,*

$$wp(B, \mathit{true}) = \mathit{true}$$

for $B \in \{\mathit{skip}, v := e, g \Rightarrow B_1, B_1\ ;\ B_2, B_1\ [\,]\ B_2\}$*.*

Proof:

$$
\begin{aligned}
& wp(B, true) && \{\text{Theorem 5.1}\}\\
&= \neg\phi(B \;;\; \neg true) && \{\text{simplify}\}\\
&= \neg\phi(B \;;\; \mathit{false}) && \{\text{Fig. 5.2: } \phi(B_1 \;;\; B_2)\}\\
&= \neg\Big(\exists\, \overline{v_0} : \big(\phi(B)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \phi(\mathit{false})[\overline{v} \leftarrow \overline{v_0}]\big)\Big) && \{\text{simplify}\}\\
&= \neg\Big(\exists\, \overline{v_0} : \big(\phi(B)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \mathit{false}\big)\Big) && \{\text{simplify}\}\\
&= \neg\big(\exists\, \overline{v_0} : (\mathit{false})\big) && \{\text{simplify}\}\\
&= \neg(\mathit{false}) \quad = \quad true && \square
\end{aligned}
$$

Having clarified the relation between the classical weakest pre-condition semantics and our predicative semantics of action systems, we conclude this chapter by giving an overview of extensions of action systems and related formalisms.

5.4 Extensions and Related Formalisms

Various extensions of the classical action system formalism have been developed. In the following, we describe an extension regarding object-orientation: Object-Oriented Action Systems (OOASs). It is relevant for this work as it is used as an intermediate language in the MoMuT::UML tool chain for transforming UML state machines into action systems as described in the previous section. Subsequently, we briefly discuss extensions of action systems allowing for modelling of hybrid systems. Furthermore, there exist many formalisms that are closely related to action systems. In the following, we shortly review Action-Oberon, Event-B, UNITY, and TLA.

5.4.1 Object-Oriented Action Systems

Bonsangue et al. [45] integrated object-orientation into action systems resulting in the *OOAS* formalism. OOASs model classes. Instances thereof (objects) can be dynamically created, may be active, and distributed. Hence, several objects can be executed in parallel, where objects interact via remote procedure calls and shared variables.

An OOAS consists of a finite set of classes. A class defines data and behaviour. The data part comprises attributes, which are divided into *shared attributes* and *local attributes*, as well as *object variables*. Shared attributes may be used by all active objects – also by objects that are instances of another class. Local attributes can only be used by one instance of the class. Object variables are a special kind of local attributes. They contain names of objects used for calling methods thereof. It is assumed that the set of shared attributes, the set of local attributes, and the set of object variables are pairwise disjoint. The behavioural part of a class defines *methods*, *procedures*, and *actions*. Methods are procedures of an instance of the class that can be called by the object itself or by other objects. In contrast, a procedure is local to the object and can only be called by the object itself. Finally, the actions of a class are defined and composed in a do-od block like in classical action systems (cf. Definition 5.1). The do-od block is executed whenever an instance of the class becomes activated. It can refer to all shared attributes, its own local attributes, as well as to its own object variables. Furthermore, it may call its own procedures as well as methods of its own and of other objects. Hence, communication between objects takes place via shared variables and via remote method calls.

Some classes of an OOAS are marked to be *roots*. The execution of an OOAS starts by instantiating one object of each of these root classes. Each object iterates over its do-od block and non-deterministically chooses an enabled action for execution. Actions may be executed in parallel if they do not rely on shared data or if they operate on disjoint sets of shared data. They can also create and kill other objects. OOASs can be composed in a similar manner as original action systems (cf. Definition 5.2).

OOASs can be transformed into regular action systems. This translation relates methods to exportable procedures, shared attributes to shared variables, local attributes and object variables to internal variables, and classes to entire action systems. Furthermore, object constructors correspond to initialisation actions. A collection of classes is translated into a parallel composition of action systems. This relation between OOASs and action systems allows to reuse existing theories for action systems. For example, refinement can be used to model class inheritance.

The OOAS formalism used as an intermediate language in the MoMuT::UML tool chain (cf. Section 1.5.1) has been described in a conference paper [140] and in a manual for the *Argos* tool [188]. Argos is a predecessor of MoMuT::UML's OOAS2AS component and implements the transformation of these OOASs into the variant of action systems that is used in this work (cf. previous section). MoMuT::UML's OOASs are based on the OOAS formalism of Bonsangue et al. [45] as described above. However, MoMuT::UML's OOAS formalism has been enriched by the labelling of actions and by the possibility to prioritise objects of a particular class with respect to objects of another class. Objects of one class are assumed to have the same priority. The desired priorities between objects of different classes are stated in the so-called *system assembling block*. This is also the reason why classical action systems have been extended by an prioritising composition operator [181] (cf. Section 5.2). Bonsangue et al. [45] restrict their OOASs to be a finite set of classes. This also holds for MoMuT::UML's OOASs, but furthermore it is assumed that also the set of objects is finite. This means that object instantiation is only allowed during the initialisation of the variables and permits an easier check for finiteness. The mapping from MoMuT::UML's OOASs to our complex action systems is based on the transformation described by Bonsangue et al. [45]. The basic idea is to create one action system per object and subsequently perform a composition of all action systems either by non-deterministic choice or by prioritising composition as specified in the system assembling block. Note that in this way, the result is one single action system – not a composition of several action systems.

Other approaches for incorporating object-orientation into formalisms similar to action systems include the *DisCo* specification language [127] and *Action-Oberon*, which is presented later in this section.

5.4.2 Action Systems for Hybrid System Modelling

For the modelling of hybrid systems, i.e., systems showing discrete and continuous behaviour, several extensions of action systems have been developed. *Continuous action systems* [30] use continuous functions as values for variables and have an implicit variable *now* to represent the current time starting at zero. Similarly, *hybrid action systems* [174] introduce differential actions to model continuous behaviour. They describe how the values of variables evolve starting at their initial values. When the evolution terminates, the variables are fixed to their reached values. Inspired by hybrid action systems, *qualitative action systems* [11] have been introduced. They are more abstract than continuous and hybrid action systems. They replace quantitative, differential actions by qualitative actions relying on the ideas of qualitative reasoning [144]. Only the piecewise monotonic behaviour of functions is considered, i.e., if they are increasing, steady, or decreasing. Furthermore, the domain and range of functions are abstracted to points and intervals between them.

5.4.3 Action-Oberon

The programming language *Oberon-2* [161] is the successor of *Modula-2* [206], which emerged from *Pascal* [205]. *Action-Oberon* by Back et al. [27] extends Oberon-2 with actions and guarded procedures to express action systems. An action system in Action-Oberon is a module. Just like in original action systems, the actions are iterated in a loop and enabled actions are chosen non-deterministically. As a difference to action systems, there is a way to control this selection of actions. The execution environment allows to write an own scheduling algorithm or to manually choose the next action. Oberon-2 supports object-orientation. In Action-Oberon, object-orientation can also be expressed via so-called type-bound actions.

5.4.4 Event-B

The formal method *B* [1] has recently adopted the action-system style in the form of *Event-B* [2]. An Event-B specification consists of two parts: contexts and machines. Contexts describe the static part of the model. Roughly speaking, they contain type definitions including constants, axioms, and theorems. Machines represent the dynamic part, i.e., they describe the behaviour of Event-B models. They correspond to action systems by containing variables, which define the system state, and events. Note that in Event-B, actions as known from action systems are named *events*, while the term *action* is used for the body of an event. Events may have parameters. In contrast to action systems, there is no possibility to differentiate between different cases of an action, i.e., there is no possibility to react differently to individual guards. Hence, the distinction of cases must be expressed by splitting the event into several events. As already pointed out, the body of an event is called *action* in Event-B. It describes the state update the event performs. In contrast to our form of action systems, it only allows assignments to variables, but no nested guarded commands, non-deterministic choice, sequential composition, etc. All assignments are performed in parallel. However, the assignments may not only be deterministic as in our case, but can also be non-deterministic. Event-B makes heavy use of sets. Hence, one way for expressing a non-deterministic assignment is to assign an arbitrary element of a set. Another way is to assign values fulfilling a predicate. Event-B machines also comprise invariants that have to be ensured by the events. This was not included in the original action system formalism. The initial states are specified via a dedicated initialisation event. Event-B states proof obligations, which must be proven to show that the defined properties hold or that refinement between specifications holds. Event-B has tool support in the form of the *Rodin* Eclipse-plugin [3]. It provides an Integrated Development Environment (IDE) for Event-B including support for refinement and mathematical proofs. Furthermore, *ProB* [152] is an animator and model checker for the B-method (including Event-B). It can also be used for test case generation [203]. Two kinds of test purposes may be specified to guide test case generation: (1) either a predicate that has to be fulfilled in the end state of the test cases (with limited length), or (2) a certain operation that has to be covered by the test cases. In this way, transition coverage can be specified as a test goal. Furthermore, the animation feature of ProB has been used indirectly for test case generation using test scenarios [156].

5.4.5 UNITY

UNITY is a programming and proof theory by Chandy and Misra [65]. The main motivation was to make the design of programs independent of the architecture. *UNITY* programs are very similar to action systems. They include non-determinism and have no explicit means to steer the control flow. They have a system state that is updated by possibly parallel assignments. An initial state may be defined for some or all variables. Assignments may also be expressed in a quantified way, e.g., instead of the parallel assignment of array entries $A[0], A[1], ..., A[n] := B[0], B[1], ..., B[n]$, it is possible to use a quantified assignment with an index variable i: $\langle || \; i : 0 \leq i \leq n :: A[i] := B[i] \rangle$. Furthermore, an assignment may be conditional, e.g., $x := 0 \;\; \mathit{if} \;\; y < 0$. The execution of a *UNITY* program starts in an initial state (there can be more than one). In each execution step, a statement of the program body is selected non-deterministically. An assumption of *UNITY* is fairness in statement selection: in an execution with an infinite number of steps, each statement is executed infinitely often. Furthermore, in contrast to original action systems, *UNITY* supports invariants.

5.4.6 Temporal Logic of Actions (TLA)

Leslie Lamport combined two logics in TLA [146]: a logic of actions and a standard temporal logic. TLA uses familiar mathematical operators (e.g. logical conjunction) and introduces just three new operators, mainly for expressing temporal properties. Like action systems, TLA is state-based. The first additional operator is the prime operator '. Actions are relations between old states (unprimed variables) and new states (primed variables). This corresponds more or less directly to our predicative semantics for action systems. Like in the action system formalism, actions are considered to be atomic. What makes the main difference to action systems is the temporal component. The $\Box$ operator is a unary operator expressing that its operand has to hold *always*. The execution of a TLA specification is considered to be a sequence of steps resulting in new states, i.e., a sequence of states. Hence, $\Box F$ expresses that F is true in each state, i.e., at all times. In combination with negation, it allows to express safety as well as liveness properties. Furthermore, TLA presumes stuttering, which does not change the state. The reason is to adjust to different sampling rates such that more fine-grained specifications can refine more coarse-grained ones. To prevent infinite stuttering, a liveness property has to be added. This is accomplished by a fairness condition that states that if an action is possible, it must be executed eventually. Finally, the $\exists$ operator has been added to support the hiding of variables.

5.4.7 Circus

A further language for modelling concurrent and reactive systems is called *Circus*. Like the refinement calculus for action systems, there is also a theory of refinement for Circus. Circus combines Communicating Sequential Processes (CSP) [118, 175] and the *Z* notation [207]. Therefore, it is possible to describe both state information and communication aspects with Circus. Circus specifications consist of processes, which are related to action systems. Processes comprise a state, actions, and a distinguished nameless main action. The state of a process is defined as a Z schema and is internal to the process. Hence, unlike in action systems variables cannot be used for communication. Instead, Circus offers channels for inter-process communication. In action systems, actions are (nested) guarded commands. This also holds for the actions of a Circus process. However, more constructs may be used in Circus actions: Z schema expressions or invocations of other actions possibly combined via CSP operators. These CSP operators may also be used in the main action of a process to compose actions. In this way, the main action defines the overall behaviour of the process and can be seen as the counterpart of an action system's do-od block. However, Circus permits more control over the overall process behaviour. In action systems, the do-od block is an iteration over the enabled actions. Circus allows to model this via recursion, but does not necessarily require any iteration. Furthermore, original action systems non-deterministically choose between the actions. Circus allows for a wide range of CSP operators: sequential composition, parallel execution with specified synchronisation channels, interleaving composition, event hiding, external/internal choice, etc. These operators can also be used to construct a process from other processes. What is also noticeable is that not only actions but also processes may have parameters in Circus. In contrast, an action system usually does not have parameters. Note that the semantics of Circus is based on UTP [119], which also inspired our predicative semantics for action systems.

6 Refinement Checking of Action Systems

Parts of this chapter have been published in MBT 2012 [14], CSTVA 2012 [15], and QSIC 2012 [13].

In Chapter 4, we presented our model-based mutation testing approach, which depends on the used modelling formalism and a suitable conformance relation. The conformance relation determines whether a mutated model conforms to the original model. We discussed conformance relations in Chapter 3. In particular, we defined relational refinement for predicative semantics (Definition 3.11). In Chapter 5, we introduced action systems to model reactive systems and defined a relational predicative semantics for action systems. In this chapter, we link the individual components to implement a model-based mutation testing approach for action systems using the defined refinement relation. We start by relating refinement to model-based mutation testing. Subsequently, we focus on refinement of action systems. We describe our refinement checking approach and point out pitfalls that have to be considered. Finally, we illustrate our refinement checking process on the Car Alarm System (CAS) and report on experimental results for this CAS and for the particle counter use case.

6.1 Model-Based Mutation Testing using Refinement

The key idea in model-based mutation testing is to create a test case whenever a mutated model does not conform to the original model. The resulting test cases witness non-conformance, i.e., they are able to distinguish the mutated from the original model.

Aichernig and He [12] developed a mutation testing theory based on our notion of refinement (Definition 3.11): using refinement as a conformance relation, a test case is generated whenever a mutated model M^M does not refine an original model M^O, i.e., if $M^O \not\sqsubseteq M^M$. Hence, test cases are based on counterexamples to refinement. From Definition 3.11, it follows that such counterexamples exist if and only if implication does not hold.

Proposition 6.1 (Non-Refinement)

$$M^O \not\sqsubseteq M^M \quad \Leftrightarrow \quad \exists\, \overline{v}, \overline{v}' \;:\; M^M(\overline{v}, \overline{v}') \wedge \neg M^O(\overline{v}, \overline{v}')$$

Proof:

$$\begin{aligned}
& M^M \not\sqsubseteq M^O && \{\text{Definition 3.11}\}\\
&= \neg(\forall\, \overline{v}, \overline{v}' : M^M(\overline{v}, \overline{v}') \;\Rightarrow\; M^O(\overline{v}, \overline{v}')) && \{\text{first-order predicate calculus}\}\\
&= \exists\, \overline{v}, \overline{v}' : \neg(M^M(\overline{v}, \overline{v}') \;\Rightarrow\; M^O(\overline{v}, \overline{v}')) && \{\text{definition of implication}\}\\
&= \exists\, \overline{v}, \overline{v}' : \neg(\neg M^M(\overline{v}, \overline{v}') \vee M^O(\overline{v}, \overline{v}')) && \{\text{De Morgan}\}\\
&= \exists\, \overline{v}, \overline{v}' \;:\; M^M(\overline{v}, \overline{v}') \wedge \neg M^O(\overline{v}, \overline{v}') && \Box
\end{aligned}$$

Proposition 6.1 expresses non-refinement. It states that there are observations $\overline{v}, \overline{v}'$ in the mutant M^M that are not allowed by the original model M^O. We call a state, i.e., a valuation of all variables, *unsafe* if it enables unspecified observations.

Definition 6.1 (Unsafe State)
A pre-state $\overline{u}$ is called unsafe if it shows wrong (not conforming) behaviour in a mutated model M^M with respect to an original model M^O. Formally, we have:

$$\overline{u} \in \{\overline{v} \mid \exists\, \overline{v}' : M^M(\overline{v}, \overline{v}') \wedge \neg M^O(\overline{v}, \overline{v}')\}$$

An unsafe state can lead to an incorrect next state. Model-based mutation testing aims at generating test cases that cover such unsafe states. Hence, the fault-based testing criteria are based on the notion of unsafe states.

6.2 Non-Refinement of Action Systems

In the previous section, we have introduced non-refinement as a general criterion for identifying unsafe states, which are crucial for mutation-based test case generation. In the following, we deal with the special case of action systems. More precisely, we concentrate on plain action systems as presented in Section 5.2.2.

Our predicative semantics (cf. Figure 5.2) defines the observations in our action system language as the event traces and the system states before $(\overline{v}, tr)$ and after one execution $(\overline{v}', tr')$ of the do-od block. A mutated action system AS^M refines its original version AS^O if and only if all observations possible in the mutant are allowed by the original. Hence, our notion of refinement is based on both, event traces and states. However, in an action system not all states are reachable from the initial state. Therefore, reachability has to be taken into account. We reduce the general refinement problem of action systems to a step-wise simulation problem only considering the execution of the do-od block from reachable states.

Definition 6.2 (Refinement of Action Systems)
Let AS^O and AS^M be two action systems with $P^O(\overline{v}, \overline{v}', tr, tr')$ and $P^M(\overline{v}, \overline{v}', tr, tr')$ being the semantics of their corresponding do-od blocks. Furthermore, we assume the existence of a function *"reach"* that returns the set of reachable states for a given trace in an action system. Then

$$AS^O \sqsubseteq AS^M \;=_{df}\; \forall\, \overline{v}, \overline{v}', tr, tr' : ((\overline{v} \in reach(AS^O, tr) \wedge P^M(\overline{v}, \overline{v}', tr, tr')) \Rightarrow P^O(\overline{v}, \overline{v}', tr, tr'))$$

This definition is different to Back's original refinement definition for action systems, which is solely based on state traces [31]. Here, also the possible event traces are taken into account. Hence, also the action labels have to be refined.

Negating this refinement definition and considering the fact that the do-od block $P(\overline{v}, \overline{v}', tr, tr')$ of a plain action system is a non-deterministic choice of actions $A_i(\overline{v}, \overline{v}', tr, tr')$ (cf. Section 5.2.2) leads to the following formula.

Lemma 6.1

$$AS^O \not\sqsubseteq AS^M \;\Leftrightarrow\; \exists\, \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr) \wedge$$
$$(A_1^M(\overline{v}, \overline{v}', tr, tr') \vee \cdots \vee A_n^M(\overline{v}, \overline{v}', tr, tr')) \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \cdots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr'))$$

Proof:

$AS^O \not\sqsubseteq AS^M$ {Definition 6.2}

$= \neg(\forall\, \overline{v}, \overline{v}', tr, tr' : ((\overline{v} \in reach(AS^O, tr) \wedge P^M(\overline{v}, \overline{v}', tr, tr')) \Rightarrow P^O(\overline{v}, \overline{v}', tr, tr')))$

{first-order predicate calculus}

$= \exists\, \overline{v}, \overline{v}', tr, tr' : \neg((\overline{v} \in reach(AS^O, tr) \wedge P^M(\overline{v}, \overline{v}', tr, tr')) \Rightarrow P^O(\overline{v}, \overline{v}', tr, tr'))$

{definition of implication}

$= \exists\, \overline{v}, \overline{v}', tr, tr' : \neg(\neg(\overline{v} \in reach(AS^O, tr) \wedge P^M(\overline{v}, \overline{v}', tr, tr')) \vee P^O(\overline{v}, \overline{v}', tr, tr'))$

{first-order predicate calculus}

$= \exists\, \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr) \wedge P^M(\overline{v}, \overline{v}', tr, tr')) \wedge \neg P^O(\overline{v}, \overline{v}', tr, tr'))$

$\{P^M(\overline{v}, \overline{v}', tr, tr') \ =_{df}\ (A_1^M(\overline{v}, \overline{v}', tr, tr') \vee \dots \vee A_n^M(\overline{v}, \overline{v}', tr, tr'))\}$

$\{P^O(\overline{v}, \overline{v}', tr, tr') \ =_{df}\ (A_1^O(\overline{v}, \overline{v}', tr, tr') \vee \dots \vee A_m^O(\overline{v}, \overline{v}', tr, tr'))\}$

$= \exists\, \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr)$

$\wedge\, (A_1^M(\overline{v}, \overline{v}', tr, tr') \vee \dots \vee A_n^M(\overline{v}, \overline{v}', tr, tr')) \wedge \neg(A_1^O(\overline{v}, \overline{v}', tr, tr') \vee \dots \vee A_m^O(\overline{v}, \overline{v}', tr, tr')))$

{De Morgan}

$= \exists\, \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr)$

$\wedge\, (A_1^M(\overline{v}, \overline{v}', tr, tr') \vee \dots \vee A_n^M(\overline{v}, \overline{v}', tr, tr')) \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \dots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr'))$ □

By application of the distributive law, disjunction becomes the outermost operator and the following set of constraints for detecting non-refinement of action systems is obtained.

Theorem 6.1 (Non-refinement of Action Systems)
A mutated action system AS^M does not refine its original AS^O iff any action $A_i^M(\overline{v}, \overline{v}', tr, tr')$ of the mutant shows trace or state-behaviour that is not possible in the original action system:

$$AS^O \not\sqsubseteq AS^M \quad \Leftrightarrow \quad \bigvee_{i=1}^{n} \exists\, \overline{v}, \overline{v}', tr, tr' :$$

$$(\overline{v} \in reach(AS^O, tr) \wedge\ A_i^M(\overline{v}, \overline{v}', tr, tr') \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge ... \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr'))$$

Proof:

$AS^O \not\sqsubseteq AS^M$ {Lemma 6.1}

$= \exists\, \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr) \wedge$

$(A_1^M(\overline{v}, \overline{v}', tr, tr') \vee \dots \vee A_n^M(\overline{v}, \overline{v}', tr, tr')) \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \dots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr'))$

{distributive law}

$= \exists\, \overline{v}, \overline{v}', tr, tr' :$

$((\overline{v} \in reach(AS^O, tr) \wedge A_1^M(\overline{v}, \overline{v}', tr, tr') \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \dots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr')) \vee$

$\dots \vee$

$(\overline{v} \in reach(AS^O, tr) \wedge A_n^M(\overline{v}, \overline{v}', tr, tr') \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \dots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr')))$

{first-order predicate calculus}

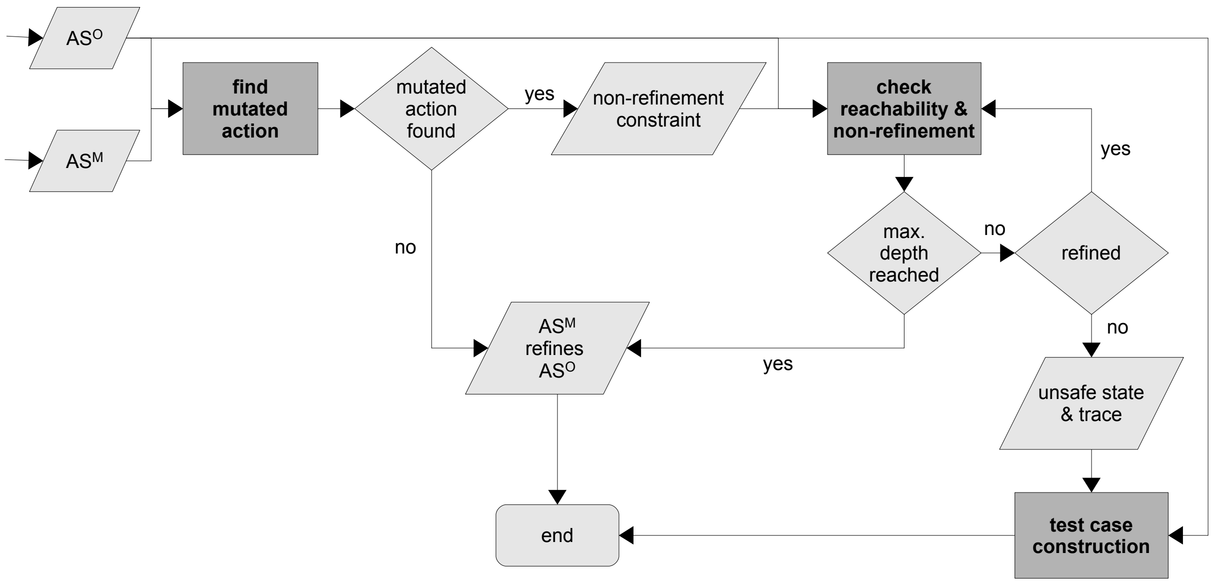


Figure 6.1: Process for test case generation via a refinement check.

$$= (\exists\, \overline{v}, \overline{v}', tr, tr' :$$

$$(\overline{v} \in reach(AS^O, tr) \wedge A_1^M(\overline{v}, \overline{v}', tr, tr') \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \cdots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr'))) \vee$$

$$\ldots \vee$$

$$(\exists\, \overline{v}, \overline{v}', tr, tr' :$$

$$(\overline{v} \in reach(AS^O, tr) \wedge A_n^M(\overline{v}, \overline{v}', tr, tr') \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge \cdots \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr')))$$

$$\{\text{notation}\}$$

$$= \bigvee_{i=1}^{n} \exists\, \overline{v}, \overline{v}', tr, tr' :$$

$$(\overline{v} \in reach(AS^O, tr) \wedge A_i^M(\overline{v}, \overline{v}', tr, tr') \wedge \neg A_1^O(\overline{v}, \overline{v}', tr, tr') \wedge ... \wedge \neg A_m^O(\overline{v}, \overline{v}', tr, tr')) \qquad \square$$

In the next section, we show how Theorem 6.1 is applied in our refinement checking process.

6.3 Refinement Checking of Action Systems

Our refinement checking approach for action systems focuses on mutation testing, i.e., the inputs are an original action system and a mutated version, which differs from the original only by small syntactical changes. For now, we concentrate on plain action system as defined in Section 5.2.2. For the integration into the MoMuT::UML tool chain (Chapter 9), we will adapt our approach in order to support complex action systems introduced in Section 5.2.3. Furthermore, we check for operational refinement (cf. Definition 3.9) and do not consider data refinement (cf. Definition 3.10). Hence, if a mutation operator changes the structure of the states of an action system, i.e., removes, adds, or renames variables that constitute the state, it will be considered as not conforming. Furthermore, we assume that the testing interface is not mutated, i.e., action signatures are not mutated and no new actions are introduced by a mutation operator.

Figure 6.1 gives an overview of our approach to find an unsafe state (Definition 6.1). The inputs are an original plain action system AS^O and a mutated version AS^M. Each plain action system consists of a set of actions AS_i^O and AS_j^M respectively, which are connected via non-deterministic choice. The first step is a preprocessing activity to check for refinement quickly. It is depicted on the left-hand side of Figure 6.1 as box *find mutated action*. If there does not exist an unsafe state at this point, we cannot find any mutated action that yields non-conformance. Hence, we already know that refinement holds between the action systems. If we find an unsafe state in this phase, we cannot be sure that it is reachable from the initial state of the action system. But we know which action has been mutated and are able to construct a *non-refinement constraint*, which describes the set of all unsafe states. The next step performs a reachability analysis and uses the non-refinement constraint to test each reached state whether it is an unsafe state. In the following, we give more details.

6.3.1 Finding a Mutated Action

The non-refinement condition presented in Theorem 6.1 is a disjunction of constraints. Each constraint deals with one action A_i^M of the mutated action system AS^M. Due to disjunction, it is sufficient to satisfy one of these sub-constraints in order to find non-refinement. We use this insight in our implementation as we perform the non-refinement check action by action (with regard to the mutant's actions A_i^M). In this section, we first concentrate on finding a possibly unreachable unsafe state. Reachability is dealt with separately in the next section.

Algorithm 6.1 gives details on the action-wise non-refinement check, which is depicted on the left-hand side of Figure 6.1 (box *find mutated action*). As inputs it takes an original action systems AS^O, a mutated version AS^M, and variables $(\overline{v}, \overline{v}', a, \overline{P})$ that will be used to represent the observations that can be made in one do-od block. It returns the name A_i^M of the action that has been mutated together with its corresponding non-refinement constraint $CS\_nonrefine$.

In Line 1, we transform the whole do-od block of the original action system AS^O into a constraint system CS_AS^O according to our predicative semantics for plain action systems (cf. black parts in Figure 5.2). We then translate one action A_i^M of the mutated action system AS^M into a constraint system $CS\_A_i^M$ (Line 3). To decide about non-refinement between two action systems (Theorem 6.1), the observations from both systems are expressed by the same alphabet $(\overline{v}, \overline{v'}, tr, tr')$. We only deal with one iteration of the do-od block and use plain action systems, which do not allow sequential compositions of actions. Hence, only one action can be executed in one iteration of the do-od block, which makes it sufficient to encode the new part of the trace with one variable a representing this action. The consecution of the executed action to the previous trace is performed externally. Therefore, our translation relies on a pre-state encoded by a variable vector $\overline{v}$, a post-state given as variable vector $\overline{v}'$, and one action variable a as well as a sequence of variables for its parameters $\overline{P}$. These variables are fixed outside of our algorithm, i.e., they are inputs for our algorithm, and are passed to the *trans* function such that the observations of the original action system and those of the mutated action systems are encoded using the same variables, i.e., CS_AS^O as well as $CS\_A_i^M$ have the same alphabet comprising $\overline{v}$, $\overline{v}'$, a, and $\overline{P}$. The set of variables $\overline{P}$ representing the parameters is used for each action to encode its parameters. As actions may have a different number of parameters, our implementation determines the size of the set $\overline{P}$ by the maximum cardinality of an action. For actions with less or no parameters at all, the spare parameter variables are constrained to a special value indicating that they are unused. Note that the action parameters $\overline{P}$ belong to the observations of both systems. Hence, they must not be existentially quantified in each individual action system as suggested in our predicative semantics for an action system (cf. Figure 5.2), which would mean that they had local scope in each system and would not be related at all.

The non-refinement constraint $CS\_nonrefine$ is the conjunction of the constraint system representing the actually mutated action ($CS\_A_i^M$) and the negated constraint system representing the original action system ($\neg CS\_AS^O$, cf. Line 4). Apart from ignoring reachability, the non-refinement constraint

Algorithm 6.1 $\mathit{findMutatedAction}(AS^O, AS^M, \overline{v}, \overline{v}', a, \overline{P}) : (A_i^M, \mathit{CS\_nonrefine})$

```
1: $\mathit{CS\_AS}^O := \mathit{trans}(AS^O, \overline{v}, \overline{v}', a, \overline{P})$
2: for all $A_i^M \in AS^M$ do
3:     $\mathit{CS\_A}_i^M := \mathit{trans}(A_i^M, \overline{v}, \overline{v'} a, \overline{P})$
4:     $\mathit{CS\_nonrefine} := \mathit{CS\_A}_i^M \wedge \neg \mathit{CS\_AS}^O$
5:     if $sat(\mathit{CS\_nonrefine})$ then
6:         return $(A_i^M, \mathit{CS\_nonrefine})$    // mutated action found
7:     end if
8: end for
9: return $(nil, \mathit{false})$    // refinement holds
```

$\mathit{CS\_A}_i^M \wedge \neg \mathit{CS\_AS}^O$ corresponds to one sub-formula of Theorem 6.1. Its alphabet is the same as for its individual components, i.e., $\overline{v}$, $\overline{v}'$, a, and $\overline{P}$.

The non-refinement constraint for the just translated action is then given to a constraint solver to check whether it is satisfiable by some $\overline{v}, \overline{v}', a, \overline{P}$ (Line 5), i.e., whether there exists an unsafe state $\overline{v}$ for AS^M and AS^O. If yes, we found the mutated action and return it together with the according non-refinement constraint $\mathit{CS\_nonrefine}$. Otherwise, the next action A_i^M is investigated (loop in Line 2). If no action leads to a satisfiable non-refinement constraint, then AS^M refines AS^O (Line 9). Algorithm 6.1 is sound for first-order mutants that only incorporate one syntactical change per mutant (cf. Definition 4.10). It aborts after finding the first action that leads to an unsafe state. Note that we do not know yet whether an unsafe state is actually reachable. For higher-order mutants with more than one syntactical change per mutant (cf. Definition 4.11), it could happen that our algorithm finds a mutated action for which no unsafe state is reachable. In this case, it is necessary to backtrack and search for another mutated action until an unsafe state is actually reachable or all actions are processed.

Example 6.1. Consider an action system comprising one state variable s of type int with a range between 0 and 4. Furthermore, assume two actions a and b. The action a has no parameters and is defined as:

$$a :: (true) => skip$$

while the action b has one parameter B and is defined as:

$$b(B) :: (B \ \#\!>= 0) => (s := B)$$

Let the do-od block of this action system be a non-deterministic choice of the two actions:

$$a \ [\,] \ b(2)$$

To demonstrate our translation, let V be the Prolog variable used to represent the pre-state variable s. Let the variable Vp encode the post-state variable s'. Let the variable A be used to encode the chosen action. Furthermore, the variable P will be used represent the parameter of action b in the resulting constraint system. Moreover, let the integer representing the action a be 1, and the integer encoding the action b be 2. The special integer indicating that a parameter is unused is represented by a special integer, e.g., -1000. The translation of action a is then:

$$A = 1 \wedge P = -1000 \wedge \ Vp = V$$

Similarly, action b is translated to:

$$A = 2 \wedge P = 2 \wedge 2 \geq 0 \wedge \ Vp = 2$$

Note that unification is used to relate the actual parameter 2 with the formal parameter B. Hence, the resulting formula directly uses the value 2. We encode that this action is named b by $A = 2$ as b is represented by the integer 2. Similarly, we state that its observable parameter P is equal to its actual parameter, i.e., 2. The whole do-od block is the disjunction of these two formulae:

$$(A = 1 \land P = -1000 \land Vp = V) \lor (A = 2 \land P = 2 \land 2 \geq 0 \land Vp = 2)$$

This formula corresponds to CS_AS^O in Algorithm 6.1.

An action of a mutated version of this action system is translated analogously. Let

$$a :: (true) => s := 0$$

be the mutated version of action a. The resulting constraint system is

$$A = 1 \land P = -1000 \land Vp = 0$$

and was denoted with $CS\_A_i^M$ in Algorithm 6.1. What is important is that the same variables (V, Vp, A, and P) are used to encode the observations. Note that this is possible as we presume that the state variables are the same in both action systems and that the testing interface is not mutated, i.e., the action signatures are not mutated and no new actions are introduced. The resulting non-refinement constraint, denoted $CS\_nonrefine$ in Algorithm 6.1, is:

$$\begin{gathered}(A = 1 \land P = -1000 \land Vp = 0) \land \\ \neg((A = 1 \land P = -1000 \land Vp = V) \lor (A = 2 \land P = 2 \land 2 \geq 0 \land Vp = 2))\end{gathered} \qquad \square$$

What needs to be stated additionally are the domains of the used variables. The range of variable A is determined by our integer encoding for the action names. In this example, it ranges from 1 to 2. The pre- and post-state variables V and Vp range from 0 to 4 as defined by type int. Finally, the parameter P may be either unused (-1000), or the constant 2. As the used constraint solver always needs restricted ranges, we constrain parameters to a given range, e.g., between -1000 and 1000. If this is not sufficient for a given action system, these values can be adjusted. Given these ranges, the non-refinement constraint is satisfiable, e.g., by $A = 1, P = -1000, V = 1, Vp = 0$. This is checked in Line 5 of Algorithm 6.1. Remember that this non-refinement constraint was built on the action a of the mutated action system. Due to its satisfiability, Algorithm 6.1 returns the action a as the mutated action and the corresponding non-refinement constraint.

Identifying the mutated action is important for performance given the used constraint solver. We experienced that solving the non-refinement constraint $CS\_nonrefine$ for one action of the mutant is by far faster than solving a non-refinement constraint encoding all actions of the mutated action system at once. Experiments showed that the latter is impractical with the used constraint solver.

6.3.2 Reaching an Unsafe State

Now we know whether there exists an unsafe state. If this is the case, we also know which action has been mutated and we have determined a non-refinement constraint that describes the set of all possible unsafe states. But we do not know yet, whether an unsafe state is actually reachable from a given initial state. It is possible that an unsafe state exists theoretically and has been found in the previous step, but that no unsafe state is reachable from the initial state of the system. In this case, the mutated action system conforms to the original, i.e., the mutant refines the specification (cf. Definition 6.2). To find out whether an unsafe state is actually reachable, we perform a state space exploration of the original action system AS^O. During this reachability analysis, each encountered state is examined if it is an

Algorithm 6.2 $\mathit{reachNonRefine}(AS^O, \mathit{CS\_nonrefine}, \overline{v}, \overline{v}', a, \overline{P}, \mathit{max}, \mathit{init}) : (\mathit{unsafe}, \mathit{trace})$

```
1: if $sat(CS\_nonrefine \wedge \overline{v} = init)$ then
2:    return $(init, [])$
3: end if
4: $Visited := \{init\}$
5: $ToExplore := enqueue((init, []), [])$
6: while $ToExplore \neq []$ do
7:    $(s_0, tr\_s_0) := head(ToExplore)$
8:    $ToExplore := dequeue(ToExplore)$
9:    if $length(tr\_s_0) < max$ then
10:      for all $(s_1, a_1, params_1) \in succ(s_0) : s_1 \notin Visited$ do
11:         $tr\_s_1 := add(tr\_s_0, a_1(\overline{P}))$
12:         if $sat(CS\_nonrefine \wedge \overline{v} = s_1)$ then
13:            return $(s_1, tr\_s_1)$   // unsafe state
14:         end if
15:         $Visited := add(s_1, Visited)$
16:         $ToExplore := enqueue((s_1, tr\_s_1), ToExplore)$
17:      end for
18:   end if
19: end while
20: return $(nil, [])$   // refinement holds
```

unsafe state. This test is realised via a constraint solver that checks whether the reached state fulfils our non-refinement constraint (see right-hand side of Figure 6.1).

The pseudo-code shown in Algorithm 6.2 gives more details on our combined reachability and non-refinement check. The algorithm requires the following inputs: (1) the original action system AS^O, (2) the constraint system $\mathit{CS\_nonrefine}$ representing the non-refinement constraint obtained from Algorithm 6.1, (3) the variables $\overline{v}$ encoding the pre-state in the non-refinement constraint, (4) the variables $\overline{v}'$ encoding the post-state, (5) the variable a encoding the action, and (6) the variables $\overline{P}$ representing the action parameters in the non-refinement constraint. Further inputs to the algorithm are (7) an integer max restricting the search depth, and (8) the initial state $init$ of the action system AS^O. The algorithm returns a pair consisting of the found unsafe state and the trace leading there.

At first (Lines 1 to 3), we check whether the initial state is already an unsafe state. That is, we call the constraint solver with the non-refinement constraint and require the pre-state variables $\overline{v}$ to be equal to the given initial state $init$ of AS^O. If these constraints are satisfiable, we detected non-refinement. We found either a state that can be reached from $init$ only in the mutant but not in the original, i.e., a valuation for $\overline{v}'$, or an action with parameters, i.e., values for a and $\overline{P}$, that is enabled at state $init$ only in the mutant but not in the original. In this case, $init$ is returned as unsafe state together with the empty trace. Otherwise, we perform a breadth-first search (Lines 6 to 19) starting at $init$. The queue $\mathit{ToExplore}$ holds the states that have been reached so far and still have to be further explored. It contains pairs consisting of the state and the shortest trace leading to this state. The set $\mathit{Visited}$ holds all states that have been reached so far and is maintained to avoid the re-exploration of states. To ensure termination, the state space is only explored up to a user-defined depth max (Line 9).

The function $succ(s_0)$ (Line 10) returns the set of all successors of state s_0. Each successor is a tuple consisting of (1) the successor state s_1, and (2) the action a_1 with (3) parameters $params_1$ leading from s_0 to s_1. The successors are calculated via the predicative semantics of our plain action systems (cf. black parts in Figure 5.2). The semantic predicates represent a constraint system encoding the transition relation of our original action system. It describes one iteration of the do-od block. Again, the observations in the form of the pre-state $\overline{v}$, the post-state $\overline{v}'$, and the action a with parameters $\overline{P}$ are encoded by using specified variables in the constraint system. For finding the successors of a given state s_0, the constraint that $\overline{v}$, which encodes the pre-state in the constraints, is equal to s_0 is added to the constraint system. We then use a constraint solver to determine valuations for the action variable a, the parameters $\overline{P}$, and the variables $\overline{v}'$ that represent the post-state. By calling the constraint solver multiple times with an extended constraint system (with the additional restriction that the next solution has to be different from the previous ones), we get all transitions that are possible from s_0. Note that we stop when the extended constraint system is unsatisfiable. In this case, there are no further solutions.

Each state s_1 that is reached in this way and has not yet been processed ($s_1 \notin \mathit{Visited}$) is checked for being an unsafe state (Line 12). This works analogously to Line 1. If an unsafe state is found, it is returned together with the trace leading there (Line 13). Otherwise, the state is included in the set of visited states (Line 15) and enqueued together with its trace for further exploration (Line 16). If no unsafe state is found up to depth max, the mutant refines the original action system and we return the pair $(nil, [])$ as a result (Line 20). Note that the recording of the trace is not encoded in the constraint systems, but is performed in Line 11 by appending the actions and parameters obtained by the solver to the trace that leads to the pre-state.

If the mutated action system does not refine the original up to the maximum search depth, our refinement check results in an unsafe state and a sequence of actions leading to this state. Based on this trace, it is possible to create a test case. This test case extraction step is already included in Figure 6.1. However, we delay its description to Chapter 8. In this chapter, we concentrate on the refinement check. In the following section, we point out pitfalls that we encountered during the implementation of our conformance check and explain our solutions for each pitfall.

6.4 Pitfalls

Implementing model-based mutation testing via refinement checking of action systems is not totally straightforward. The following pitfalls range from obvious to more subtle problems and concern different aspects of our approach as will be seen in the following.

6.4.1 Conformance Relation

The goal of model-based mutation testing is to generate test cases that are able to distinguish two systems (an original and a mutated model). For this purpose, we need to check for conformance between the mutated and the original model. Our first pitfall affects the choice of an appropriate conformance relation.

Pitfall 1. For deterministic systems, non-equivalence checking is a standard approach for finding counterexamples to conformance that allow the derivation of distinguishing test cases, e.g., [208, 173]. However, it is not suitable when non-determinism is involved (as it is the case for action systems). Just assuming the same inputs and asserting that at least one output of the mutated system differs from the output of the original one is not sufficient and leads to wrong results. □

Example 6.2. A system returns either 1 or 2 as an output regardless of the input. This can be expressed by the constraints C^O:

$$C^O = (out^O = 1 \vee out^O = 2)$$

The variable out^O represents the return value of this original system. A mutated version could return 2 or 3. The variable out^M represents the output of the mutated system in the corresponding constraints C^M:

$$C^M = (out^M = 2 \vee out^M = 3)$$

The return values out^O and out^M represent our possible observations. In order to reveal that equivalence does not hold, it is required that the observations differ:

$$C^O \wedge C^M \wedge out^M \neq out^O$$

There exist three solutions satisfying these constraints: (1) $out^M = 2, out^O = 1$, (2) $out^M = 3, out^O = 1$, and (3) $out^M = 3, out^O = 2$. Obviously, the second and the third are real counterexamples, since the mutant returns 3, which is not specified by the original system – neither in the first nor in the second branch. The first solution is not a real difference between the two systems. Value 1 is not the only specified output. Also 2 is allowed by the original. Hence, the mutant shows correct behaviour if it returns 2. □

Example 6.3. The problem becomes even more obvious if we check whether a non-deterministic specification is equivalent to itself. In this case, we should not find any counterexample for conformance, since each system conforms to itself. Again, consider the system's specification C^O from above and the very same specification as implementation. The following constraints encode non-equivalence:

$$C^O = (out^O = 1 \vee out^O = 2)$$
$$C^M = (out^M = 1 \vee out^M = 2)$$
$$C^O \wedge C^M \wedge out^M \neq out^O$$

There exist solutions for these constraints ($out^M = 2, out^O = 1$ and $out^M = 1, out^O = 2$), which is not what we expect. Hence, for non-deterministic systems, equivalence checking leads to false-positive counterexamples. □

These examples illustrate that an equivalence relation assuming the same inputs and at least one different output is not a suitable conformance relation for non-deterministic systems. We already considered conformance relations in Chapter 3, where we defined the characteristics of an equivalence relation (Definition 3.5). We also pointed out that equivalence is very strict and does not allow specifications to be more abstract than their implementations. Abstraction often involves non-determinism. We also pointed out that useful conformance relations are relations relying on some ordering from abstract to more concrete models. One of these order relations is relational refinement (Definition 3.11). Earlier in this chapter, we already showed how non-refinement expresses the main idea behind model-based mutation testing (Proposition 6.1).

Example 6.4. Reconsider the above examples. Using non-refinement as expressed in Proposition 6.1, we now get the following constraints (note that we assume the same observations, i.e., post-states represented by out^M and out^O):

$$C^M \wedge \neg C^O \wedge out^M = out^O$$

Considering C^O and C^M as stated in Example 6.2, we now get only one solution ($out^M = out^O = 3$), which is the only real counterexample. Considering C^O and C^M from Example 6.3, where C^M was actually equal to C^O, we get no solution any more. This reflects what we would naturally expect since we compared two identical systems. □

6.4.2 Semantics

In Figure 5.2, we already defined the semantics of action systems via predicates. In the following, we explain why we chose this semantics.

For encoding deterministic programs as constraints [103, 70, 208, 173], a common approach is to use Static Single Assignment (SSA) form [20]. The SSA form is an established intermediate representation for programs, where each variable is defined only once. This is accomplished by introducing a new identifier for each variable on the left-hand side of an assignment. This works fine for the positive case, i.e., for the exploration of the state space of a system, but in general entails problems when negation is required as in model-based mutation testing using refinement (cf. Proposition 6.1).

Example 6.5. Consider the following specification using sequential composition and its constraint representation via SSA form:

$$out := 1; out := out + 1$$
$$C^O = (out_1^O = 1 \wedge out_2^O = out_1^O + 1)$$

A possible implementation could be:

$$out := 2$$
$$C^M = (out^M = 2)$$

Since both systems return 2 in any case, refinement holds and we should not find a counterexample. Non-refinement is expressed by the following constraints:

$$C^M \wedge \neg C^O \wedge out^M = out_2^O$$

Unfortunately, these constraints can be satisfied by $out^M = out_1^O = out_2^O = 2$, which wrongly classifies our implementation $out := 2$ as incorrect. □

Pitfall 2. The above example demonstrates that deriving constraints via the SSA form is problematic for specifications comprising sequential compositions. It may result in false positive counterexamples – even for deterministic systems. □

The problem is that the SSA form does not reflect the semantics of sequential composition in a completely correct way. During the transformation of a program into SSA form, a new intermediate variable is introduced for each variable occurring as left-hand side of an assignment. As in Example 6.5, $out := 1; out := out + 1$ is transformed into $out_1 = 1 \wedge out_2 = out_1 + 1$. In this way, an intermediate variable out_1 is introduced as free, i.e., observable, variable and causes a false counterexample as our implementation did not show such an intermediate observation. Note that we are only interested in observing final states whereas intermediate states shall be hidden. Our relational predicative semantics for action systems (Figure 5.2) correctly expresses this by existential quantification of intermediate variables, i.e., they become bound variables.

Example 6.6. Reconsider Example 6.5, where $out := 1; out := out + 1$ served as specification and $out := 2$ as implementation. Using our predicative semantics, the specification is represented as

$$C^O = (\exists\, out_0 : out_0 = 1 \wedge out^{O'} = out_0 + 1)$$

Furthermore, the implementation is $C^M = (out^{M'} = 2)$. The constraint system expressing non-refinement is $C^M \wedge \neg C^O \wedge out^{M'} = out^{O'}$. It cannot be satisfied any more, i.e., the implementation refines the specification, as we show in the following:

$C^M \wedge \neg C^O \wedge out^{M'} = out^{O'}$ $\{$apply $C^M, C^O\}$

$out^{M'} = 2 \wedge \neg(\exists\, out_0 : out_0 = 1 \wedge out^{O'} = out_0 + 1) \wedge out^{M'} = out^{O'}$

$\{$set $out^{M'} = out^{O'} = 2$, $out_0 = 1$, otherwise contradiction$\}$

$out^{M'} = 2 \wedge \neg(\exists\, out_0 : 1 = 1 \wedge 2 = 1 + 1) \wedge out^{M'} = out^{O'}$ $\{$first-order predicate calculus$\}$

$out^{M'} = 2 \wedge \neg(true) \wedge out^{M'} = out^{O'}$ $\{$first-order predicate calculus$\}$

$false$ □

The semantics via SSA form and our predicative semantics differ in case of negation. In the positive case, the SSA form of a sequential composition $B_1\,;\,B_2$ corresponds to the predicates $p_1(\overline{v}, \overline{v_0}) \wedge p_2(\overline{v_0}, \overline{v}')$ where p_1 represents the semantics of B_1 and p_2 represents the semantics of B_2. Existential quantification of $\overline{v_0}$ is done implicitly by the constraint solver. Basically, this makes it equal to our predicative semantics for sequential composition, which is $\exists\, \overline{v_0} : (p_1(\overline{v}, \overline{v_0}) \;\wedge\; p_2(\overline{v_0}, \overline{v}'))$ given that $\phi(B_1)[\overline{v}' \leftarrow \overline{v_0}] = p_1(\overline{v}, \overline{v_0})$ and $\phi(B_2)[\overline{v} \leftarrow \overline{v_0}] = p_2(\overline{v_0}, \overline{v'})$. However, if we negate our predicative semantics, we get $\neg(\exists\, \overline{v_0} : (p_1(\overline{v}, \overline{v_0}) \;\wedge\; p_2(\overline{v_0}, \overline{v}')))$. If we negate the SSA form, we get $\neg(p_1(\overline{v}, \overline{v_0}) \wedge p_2(\overline{v_0}, \overline{v}'))$. Again, the constraint solver implicitly existentially quantifies all variables and we get $\exists \overline{v}, \overline{v_0}, \overline{v}' : (\neg p_1(\overline{v}, \overline{v_0}) \wedge p_2(\overline{v_0}, \overline{v}'))$, which is wrong as the existential quantification is not negated. Hence the source of Pitfall 2 is a wrongly placed existential quantification. This leads us directly to the next pitfall.

Pitfall 3. Our predicative semantics leads to the following constraints for a sequential composition $B_1\,;\,B_2$ that is negated:

$$\neg(\exists\, \overline{v_0} : (\phi(B_1)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \phi(B_2)[\overline{v} \leftarrow \overline{v_0}]))$$

By resolving negation, we get

$$\forall\, \overline{v_0} : (\neg(\phi(B_1)[\overline{v}' \leftarrow \overline{v_0}] \;\wedge\; \phi(B_2)[\overline{v} \leftarrow \overline{v_0}]))$$

This constraint system expresses exactly what is intended, but uses universal quantification. Universal quantification can only be expressed in quantified constraint satisfaction problems [101], which are not supported by common constraint solvers. □

A possible solution to this problem is the so-called one-point rule.

Definition 6.3 (One-Point Rule)
Given a variable x, an expression e, and a predicate P over variable x. Given that x does not occur free in e, then:

$$(\exists\, x : x = e \wedge P(x)) \Leftrightarrow P(e)$$

This means that if the variable is fixed to one value, it is possible to substitute the value for the variable and eliminate existential quantification.

Example 6.7. Consider the formula $\exists\, x : x = 2y \wedge x > 0$. By application of the one-point rule where $P(x) = (x > 0)$, we can eliminate the existential quantification. This yields $2y > 0$. □

Pitfall 4. The application of the one-point rule is only possible if the left-hand side of a sequential composition is deterministic, i.e., it if binds a variable to one value. This is the case for assignments. Nevertheless, constructs like $(out := 1 \,[]\, out := 2) \; ; \; out := out + 1$ are possible. Its constraint representation is $\exists\, out_0 : ((out_0 = 1 \vee out_0 = 2) \wedge out' = out_0 + 1)$. In this case, the left-hand side of the sequential composition is not deterministic and as a consequence we cannot substitute since we do not know which value will be assigned to out. □

We can avoid such problems by introducing a normal form which requires that non-deterministic choice is always the outermost operator and not allowed in nested expressions. In this way, the left-hand side of a sequential composition is always deterministic and existential quantification can be eliminated. Given that sequential composition corresponds to conjunction and non-deterministic choice corresponds to disjunction, this required normal form can be related to the disjunctive normal form (DNF) in predicate logic. Hence, we can automatically rewrite each action system into this normal form.

Example 6.8. Reconsider the example used in Pitfall 4, i.e., $(out := 1 \,[]\, out := 2) \; ; \; out := out + 1$. By normalising this statement, we transform it into:

$$(out := 1 \; ; \; out := out + 1) \,[]\, (out := 2 \; ; \; out := out + 1)$$

The constraint representation of this normalised form is:

$$(\exists\, out_0 : out_0 = 1 \wedge out' = out_0 + 1) \vee (\exists\, out_0 : out_0 = 2 \wedge out' = out_0 + 1)$$

Here, the one-point rule is applicable and yields the constraints $(out' = 1 + 1) \vee (out' = 2 + 1)$. □

6.4.3 Constraint Logic Programming

As already presented in Figure 5.2, the semantics of an action also comprises its action label and optional parameters. Hence, we also have to encode action names (labels) and their parameters. At the moment we use SICStus Prolog's built-in constraint solver [62]. As it only supports integers, each action and its parameters has to be associated with an integer. Richer data structures are supported by Constraint Logic Programming (CLP), which combines logic programming with constraint solving. In this way, it becomes possible to encode actions and their parameters as Prolog terms and use unification to compare them. This facilitates the treatment of actions, but also leads us to another pitfall.

Pitfall 5. All operators dealing with operands that contain Prolog clauses must be Prolog operators. Operators from the constraint solver will not work on Prolog clauses. Prolog's operator for conjunction is a comma ',' and Prolog's operator for negation is '\+'. Hence, our constraints to find a counterexample (cf. Proposition 6.1)

$$\exists\, \overline{v}, \overline{v}' : \; M^M(\overline{v}, \overline{v}') \wedge \neg M^O(\overline{v}, \overline{v}')$$

would have to be rewritten to

$$\exists\, \overline{v}, \overline{v}' : \; M^M(\overline{v}, \overline{v}') \; , \; \backslash + \; M^O(\overline{v}, \overline{v}')$$

Unfortunately, Prolog's negation is not equivalent to logical negation – as is well known. Prolog implements negation as failure, i.e., $\backslash + P$ means that *P is not provable*, whereas $\neg P$ means *P is not true*. For a more detailed explanation, we refer to Sterling and Shapiro's comprehensive book on Prolog [186]. □

Example 6.9. Again, consider the specifications introduced in Example 6.2. From Example 6.4, we know the following counterexample: $out^M = out^O = 3$. If we use Prolog's negation as described above, we get the following CLP problem:

$$out^M = out^O, \; (out^M = 2 \vee out^M = 3), \; \backslash + (out^O = 1 \vee out^O = 2)$$

```
 1  var([f, s], bool).
 2  state_def([f, s]).
 3  init([false, false]).
 4  as :-
 5    actions (
 6       'AlarmOn'::(true) => (
 7          ((f #= false => f := true)    % mutation: ((f #= false => f := false)
 8          []
 9            (s #= false => s := true))
10          ;
11          ((f #= false => f := true)
12          []
13            (s #= false => s := true))
14       )
15    ),
16    dood ('AlarmOn').
```

Listing 6.1: Code snippet of an action system modelling a variant of the CAS.

Given that the domains for all variables range from 1 to 3, Prolog's answer for this query is *no*, i.e., it cannot find a counterexample. The reason is that the variables out^M and out^O are not instantiated before the negated term. They are just fixed to be 2 or 3. In this case, Prolog can prove $out^O = 1 \vee out^O = 2$ as the right-hand side of the disjunction is true for $out^O = 2$. As the goal $out^O = 1 \vee out^O = 2$ is provable, the negated goal fails. Hence, the whole CLP goal fails and no counterexample is found. □

The problem with negation as failure could be avoided by instantiation of all variables before the call of negation. This can be established by letting the constraint solver enumerate all possible values for the variables. Prolog's backtracking would then ensure that all possibilities are tested. This solution corresponds to other techniques that use explicit enumeration. For performance reasons, we rather stick to pure constraint solving techniques and encode actions and their parameters as integers.

6.5 Illustration with the Car Alarm System

In the following we demonstrate our refinement checking approach that resolves all of the presented pitfalls using the CAS introduced in Section 1.6.1. We already showed code snippets from an action system modelling the CAS in Listing 5.1. However, for this demonstration we introduce a simpler action system, which focuses on the activation of the alarms. Furthermore, we do not consider separate events for turning the alarms on, i.e., *'FlashOn'* and *'SoundOn'*, but only use one event *'AlarmOn'* for turning on both alarms. Listing 6.1 presents code snippets of this action system. Line 1 declares two Boolean variables f and s. They are used to indicate whether the flash and sound are turned on. It is not specified in which order these two alarms are turned on. This is modelled by the sequential composition (;) of two non-deterministic choices ([]) in the *'AlarmOn'* action. Lines 7 to 13 non-deterministically either activate the flash lights (f) or the sound (s). Subsequently, the other alarm is turned on.

The first step is to normalise the action system depicted in Listing 6.1 to avoid Pitfall 4. Remember that this normal form requires that non-deterministic choice is always the outermost operator and not allowed in nested expressions. Normalisation has to be applied to each action's body as this is the only place where our syntax allows a combination of non-deterministic choice and sequential composition (cf. Figure 5.1). Listing 6.2 shows the normalisation of the action *AlarmOn*, which is a non-deterministic choice of four sequential compositions.

Next, the action must be encoded as constraints. By applying our predicative semantics of Figure 5.2,

```
'AlarmOn'::(true) => (
  (f #= false => f := true) ; (f #= false => f := true)
[]
  (f #= false => f := true) ; (s #= false => s := true)
[]
  (s #= false => s := true) ; (f #= false => f := true)
[]
  (s #= false => s := true) ; (s #= false => s := true))
```

Listing 6.2: Normalisation of the action 'AlarmOn' defined in Listing 6.1.

$$
\begin{gathered}
\mathit{true}\ \wedge \\
(\exists\ f_0, s_0 : (f = \mathit{false} \wedge f_0 = \mathit{true} \wedge s_0 = s \wedge f_0 = \mathit{false} \wedge f' = \mathit{true} \wedge s' = s_0) \vee \\
\exists\ f_0, s_0 : (f = \mathit{false} \wedge f_0 = \mathit{true} \wedge s_0 = s \wedge s_0 = \mathit{false} \wedge f' = f_0 \wedge s' = \mathit{true}) \vee \\
\exists\ f_0, s_0 : (s = \mathit{false} \wedge f_0 = f \wedge s_0 = \mathit{true} \wedge f_0 = \mathit{false} \wedge f' = \mathit{true} \wedge s' = s_0) \vee \\
\exists\ f_0, s_0 : (s = \mathit{false} \wedge f_0 = f \wedge s_0 = \mathit{true} \wedge s_0 = \mathit{false} \wedge f' = f_0 \wedge s' = \mathit{true})) \wedge \\
a = 1
\end{gathered}
$$

Figure 6.2: Predicative semantics for Listing 6.2.

we avoid Pitfall 2. The resulting constraints are depicted in Figure 6.2 assuming that the integer encoding for label *AlarmOn* is 1. Remember that unprimed variables belong to the pre-state and primed variables represent the post-state. Furthermore, the variable a is used to encode the executed action label (cf. Equation 6.6) and there are no parameters that need to be considered. To avoid Pitfall 3, we apply the one-point rule to eliminate the quantifiers such that the constraints can be processed by a constraint solver. The result is depicted in Figure 6.3. By simplification, we can skip Equation 6.1. Equations 6.2 and 6.5 are part of a disjunction and eliminated since they are contradictions. Altogether, this results in the following constraints for the action *AlarmOn*:

$$
\begin{gathered}
((f = \mathit{false} \wedge s = \mathit{false} \wedge f' = \mathit{true} \wedge s' = \mathit{true}) \vee \\
(s = \mathit{false} \wedge f = \mathit{false} \wedge f' = \mathit{true} \wedge s' = \mathit{true})) \wedge a = 1
\end{gathered}
$$

Both cases of the disjunction are equivalent and may be reduced to one. We refer to the resulting constraints as C^O in the following:

$$C^O = (f = \mathit{false} \wedge s = \mathit{false} \wedge f' = \mathit{true} \wedge s' = \mathit{true} \wedge a = 1)$$

$$\mathit{true}\ \wedge \tag{6.1}$$

$$((f = \mathit{false} \wedge \mathit{true} = \mathit{false} \wedge f' = \mathit{true} \wedge s' = s) \vee \tag{6.2}$$

$$(f = \mathit{false} \wedge s = \mathit{false} \wedge f' = \mathit{true} \wedge s' = \mathit{true}) \vee \tag{6.3}$$

$$(s = \mathit{false} \wedge f = \mathit{false} \wedge f' = \mathit{true} \wedge s' = \mathit{true}) \vee \tag{6.4}$$

$$(s = \mathit{false} \wedge \mathit{true} = \mathit{false} \wedge f' = f \wedge s' = \mathit{true})) \wedge \tag{6.5}$$

$$a = 1 \tag{6.6}$$

Figure 6.3: Quantifier-free predicative semantics for Listing 6.2 obtained by applying the one-point rule to the constraints in Figure 6.2.

This expresses what was intended to be modelled: the action *AlarmOn* (encoded by 1) is executed if neither sound nor flash are activated yet and turns on both alarms. In Algorithm 6.1 this relates to Line 1, i.e., CS_AS^O in the algorithm is equal to C^O.

The comment (%) in Line 7 of Listing 6.1 represents a possible mutation of the action system. It sets the variable f to false instead of true. For this mutated action system, we derive the constraints C^M analogously to C^O:

$$
\begin{aligned}
C^M =(((&f = \mathit{false} \wedge f' = \mathit{true} \wedge s' = s) \vee && (6.7)\\
&(f = \mathit{false} \wedge s = \mathit{false} \wedge f' = \mathit{false} \wedge s' = \mathit{true}) \vee && (6.8)\\
&(s = \mathit{false} \wedge f = \mathit{false} \wedge f' = \mathit{true} \wedge s' = \mathit{true})) \wedge && (6.9)\\
&a = 1) && (6.10)
\end{aligned}
$$

This corresponds to Line 3 in Algorithm 6.1. For action systems with more than one action, Algorithm 6.1 performs this transformation for each action in the mutated action system until the non-refinement constraint can be satisfied. In our simple example, we have only one action. This action has been mutated and the non-refinement constraint $CS\_\mathit{nonrefine}$ is $C^M \wedge \neg C^O$. Given that all variables are either Booleans or integers ranging from 0 to 1, the constraints are satisfiable by two solutions:

1. $f = s = \mathit{false}, f' = \mathit{true}, s' = \mathit{false}, a = 1$ and
2. $f = s = \mathit{false}, f' = \mathit{false}, s' = \mathit{true}, a = 1$

Both solutions serve as counterexamples for refinement as they reveal wrong behaviour and are reachable. In our simple example, reachability of the unsafe state is trivial as the unsafe state is equal to the initial state. The original action system activates both alarms (flash and sound), i.e., it sets the variables f and s to true. The mutated action system can establish different behaviour. Consider Listing 6.1 again. Although the mutated action system can establish the correct post-state by first activating the sound (Line 9) and then enabling the flash lights (Line 11), it also might end up in an incorrect post-state by first executing the mutated statement of Line 7. Afterwards, both branches of the second non-deterministic choice are enabled. In case of choosing Line 11, the flash lights will be turned on, but no sound. This corresponds to the first solution from the constraint solver, which satisfies Equation 6.7 of C^M. If Line 13 is executed, the sound will be enabled, but no flash lights, which corresponds to the second solution from the constraint solver, which satisfies Equation 6.8. Note that Algorithm 6.1 does not enumerate all solutions. It only checks whether a solution exists and since this is the case, it returns *AlarmOn* as the mutated action and the non-refinement constraint $CS\_\mathit{nonrefine}$. The variable valuations $f = \mathit{false}$ and $s = \mathit{false}$ represent the pre-state, in which different post-state observations are enabled. Hence, $f = \mathit{false}, s = \mathit{false}$ represents an unsafe state (cf. Definition 6.1). As already mentioned, the reachability check described by Algorithm 6.2 is trivial for our simple example as the unsafe state is equal to the initial state.

6.6 Experimental Results

We implemented our refinement checking approach for plain action systems in SICStus Prolog. SICStus comes with an integrated constraint solver *clpfd* (Constraint Logic Programming over Finite Domains) [62], which we used.

The translation of the actions into constraints is straightforward except for the application of the one-point rule to eliminate the quantifiers. Therefore, we normalise the actions such that all branching happens at the beginning of each iteration through the do-od block. The application of the one-point rule is implemented via symbolic execution [138] during the translation of the do-od block. Note that our symbolic execution is simpler than in the general case as no branching occurs in between due to normalisation. In each guard, we replace all references to variables by their current symbolic values. Consecutive guards are combined via conjunction leading to our path condition. At each assignment, we update the symbolic value for the assigned variable. At the end of each path, the final symbolic value for each variable v_i is added to the path condition pc. Having a lookup function *symbVal* that takes a variable and returns its symbolic value, we have constraints of the form $pc \wedge v'_1 = symbVal(v'_1) \wedge ... \wedge v'_n = symbVal(v'_n) \wedge a = action\_id$ for each path through the action system. Note that the variable a encodes the action to which the path belongs via a unique integer. The parameters are constrained in the guards of the actions, i.e., in the path condition pc.

In the following, we report on experimental results with the CAS and the particle counter (cf. Section 1.6) using the above described implementation. We also conducted experiments with an early implementation that did only work for very simple action systems like the CAS model. As this implementation was an intermediate result and no general solution, we refrain from reporting results here. However, they can be found in our first workshop publication [14].

6.6.1 Car Alarm System

We used a plain action system model of the CAS that is almost the same as the action system depicted in Listing 5.1. The only difference is the modelling of time. In Listing 5.1, time is modelled as the first parameter of each action, while in the used action system, there is an additional action *after* to encode the passage of time.

We conducted our experiments for four different versions of the CAS: (1) *CAS_1*: the CAS as presented in Section 1.6.1 with parameter values 20, 30, and 270 for waiting times, (2) *CAS_10*: the CAS with parameter values multiplied by 10 (200, 300, and 2700), (3) *CAS_100*: the CAS with parameters multiplied by 100, and (4) *CAS_1000*: the CAS with parameters multiplied by 1000. These extended parameter ranges shall test the capabilities of our constraint-based approach.

As we do not have a mutation engine for action systems, we manually created first-order mutants for these four original CAS models by applying the following three mutation operators:

- *guard true*: Setting all possible guards to true resulted in 34 mutants.
- *comparison operator inversion*: The action system contains two comparison operators: equality (#=) and inequality (#\=). Inverting all possible equality operators (resulting in inequality) yielded 52 mutants. Substituting inequality by equality operators resulted in 4 mutants.
- *increment integer constant*: Incrementation of all integer constants by 1 resulted in 116 mutants. Note that at the upper bound of a domain, we applied the smallest value in the domain in order to avoid domain violations.

From these mutation operators, we obtained a total of 206 mutated action systems for each CAS version. Additionally, we also included the original action system as an equivalent mutant, i.e., we considered 207 mutants for each CAS version.

Our refinement checker yielded the following results: 30 mutants refine the original action system. The remaining 177 mutants do not refine the original. Note that the state space of this model could be fully explored: within 13 steps, i.e., 13 consecutive actions, all possible states of the system have been reached. Figure 6.4 illustrates the lengths of the traces leading to the unsafe states. Many mutations can already be identified in the initial state (trace length 0). However, there are also mutations that can only be revealed in the deepest states at depth 13.

We conducted our experiments on a MacBook Pro with an Intel i7 dual-core processor (2.8 GHz) and 8 GB RAM with a 64-bit operating system. Table 6.1 states the computation time required to check for

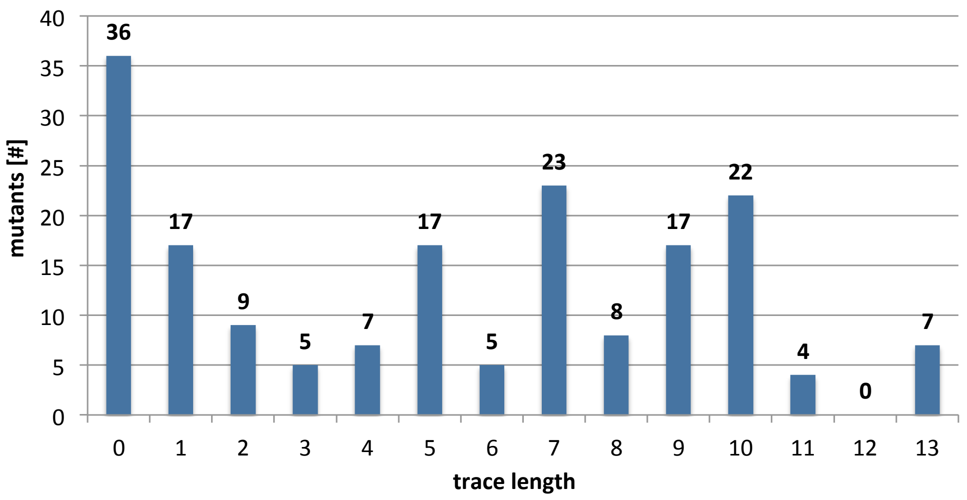


Figure 6.4: Overview of the lengths of the traces leading to the unsafe states for the CAS case study.

refinement between the original model and the 207 mutated models for each of our four CAS versions. We state the time needed to process all 207 mutants (Σ), the average time needed for one mutant (ϕ), and the maximum amount of time needed for one mutant (max). We divide the execution time into two parts: (1) the time to find the mutated action, i.e., for checking whether there possibly exists an unsafe state and which action has been mutated as described in Algorithm 6.1 (column *1: find mutated action*), and (2) the time needed for the combined reachability and non-refinement check as described in Algorithm 6.2 (column *2: reach & non-refine*). The sum thereof results in the overall execution time for refinement checking (column *total*). For CAS_1, our refinement checker needs 41 seconds to process all 207 mutants. For CAS_10, it needs already 179 seconds, then half an hour for CAS_100, and finally approximately 4 hours for CAS_1000. Note that Table 6.1 does not contain minimum values as they are 0 ms for our refinement checker, i.e., not measurable in practice. Also note that the refinement check for most mutants requires only a small amount of time. Outliers rise the average value. The arithmetic mean is between 13 seconds for CAS_1 and 3.4 hours for CAS_1000. In contrast, the median value is 0.12 to 0.15 seconds for each CAS version and 75% of the mutants can be processed in $\leq$ 0.17 to 0.23 seconds per mutant. More information can be found in Table B.1 in the appendix. It is a more detailed version of Table 6.1 as it additionally states values for the quartiles. What makes up the major part of the total computation time is finding the mutated action. It takes more than half of the time for the total refinement checking process and this already for the smallest CAS. For the largest CAS, it almost makes up 100% of the total runtime (4.2 hours for the 207 mutants). In contrast, the combined reachability and non-refinement check stays rather constant (18 - 23 seconds per CAS version for all mutants).

6.6.2 Particle Counter

We also applied our refinement checker for the particle counter use case. In particular, we used a plain action system that models the control logic of the particle counter, which was created by Stefan Tiran. Discussing the whole model goes beyond the scope of this thesis. However, to illustrate the complexity of the model, Table 6.2 relates some model metrics of this industrial use case to the corresponding values of the CAS action system models used in the previous section. Note that the metrics stated under CAS hold for each of our four models (CAS_1, CAS_10, CAS_100, and CAS_1000). The model of the particle counter consists of 26 actions and 10 state variables compared to 11 actions and 6 state variables of the CAS model. Considering the domains of these variables, this leads to a combinatorial state space of $\sim 1.6 \times 10^9$ possible state variable valuations for the particle counter compared to 1600 possible state

		1: find mutated action	2: reach & non-refine	total
CAS_1	Σ	23	18	**41**
	ϕ	0.11	0.09	0.2
	max	13	0.37	13
CAS_10	Σ	160	19	**179**
	ϕ	0.77	0.09	0.86
	max	127	0.38	127
CAS_100	Σ	32.4 min	23	**33 min**
	ϕ	9.39	0.11	9.5
	max	28 min	0.49	28 min
CAS_1000	Σ	4.2 h	18	**4.2 h**
	ϕ	73	0.09	73
	max	3.4 h	0.35	3.4 h

Table 6.1: Computation times required by our refinement checker for the four versions of the CAS. All values are given in seconds unless otherwise noted.

	particle counter	CAS
actions [#]	26	11
state variables [#]	10	6
possible states [#]	$\sim 1.6 \times 10^9$	1600
reachable states [#]	1725	21
required exploration depth	28	13
LOC	~ 390	~ 160

Table 6.2: Metrics describing the test models of the particle counter and the CAS.

variable valuations for the CAS. However, only 1725 of them are actually reachable compared to 21 reachable states of the CAS. Since our search is bounded, the chosen exploration depth is important. In order to reach all possible states in the particle counter model, an exploration depth of 28 steps is required; for the CAS it is 13. We measure the exploration depth as the number of events in a trace. For these models, a full state space exploration is possible and was performed for our experiments. However, note that if the used model is too complex and the state space cannot be fully explored, we might not detect mutations that can only be observed deep in the system. This is a general problem of bounded techniques, e.g., bounded model checking [69]. Note that Ulysses [10], the existing test case generation backend for MoMuT::UML (Section 1.5.1) also performs a bounded ioco check up to a given depth limit.

From the model described above we again generated first-order mutants. We applied the following mutation operators: setting guards to true resulted in 101 mutants, swapping equal and unequal operators resulted in 249 mutants, and incrementing integer constants by one resulted in 322 mutants. In total, we obtained 672 mutants. We started the refinement check up to depth 28 for all 672 mutants. However, only 5 mutated models could be processed within 6 hours, when we aborted due to the poor progress. The refinement checks of 4 mutants could be finished. Thereof, one mutant refines the original action system. The other 3 mutants do not refine the original. For one mutant the initial state is unsafe. The other two mutants have unsafe states at depth 5.

We conducted these experiments on the same machine we already used for the CAS experiments. Hence, the runtimes are comparable between the two case studies. As in the CAS case study (cf. Table 6.1), the vast majority of the overall computation time is needed for finding the mutated action. Only 41 seconds of the 6 hours were used for the combined reachability and refinement check. The search for the mutated action requires to check for satisfiability of the non-refinement constraint without a specified pre-state, i.e., the constraint solver has also to search for a variable valuation for the pre-state. In contrast, during our reachability analysis, the non-refinement constraint is always solved with respect to a given pre-state. To have a specified pre-state seems to immensely simplify the search for the other variables (action, parameters, and the post-state variables) for the used constraint solver.

In the next chapter, we present optimisations that allow us to cope with more complex models like the one for the particle counter.

7 Efficiency in Refinement Checking

Parts of this chapter have been published in QSIC 2012 [13] and TAP 2013 [16]. Furthermore, parts of this chapter will appear in an issue of the SCP journal [17].

The particle counter use case demonstrated that our refinement checking implementation does not yet show a reasonable performance. In this chapter, we present four optimisation techniques and report on their effect for the CAS and the particle counter models.

7.1 Optimisation Techniques

7.1.1 Variable and Value Selection Heuristics

In the previous chapter, our implementation always used the default settings of SICStus Prolog's integrated constraint solver *Constraint Logic Programming over Finite Domains (clpfd)* [62]. Obviously, different settings should be tried to see whether the performance can be improved. We modified the search strategy of the constraint solver by trying different combinations of variable and value selection strategies. Variables are provided to the solver in a list. By default, the solver selects the next variable for assignment from left to right in the given list (*leftmost*). Note that we tried different orders of the variables in the list. We found out that the best sequence is to first state the pre-state variables, then the variables used for the action its parameters, and at last the post-state variables. Other variable selection strategies that do not depend on the order of the variables in the given list include the first-fail principle (*ff*) and the most-constrained heuristic (*ffc*). The first-fail principle selects the variable with the smallest domain. The most-constrained heuristic selects the variable that has the smallest domain and additionally the most constraints suspended on it. For value selection, the default is to try values in ascending order (*up*). The other alternative is to use descending order (*down*).

7.1.2 Mutation Detection Strategies

In the previous chapter, we described our refinement relation for action systems. In Theorem 6.1, we showed that non-refinement can be expressed by a disjunction of constraints of which each one deals with one action A_i^M of the mutated action system AS^M. Due to disjunction, it is sufficient to satisfy one of these sub-constraints in order to prove non-conformance. We tested which of these sub-constraints is satisfiable, i.e., which action has been mutated. The satisfiable sub-constraint was referred to as non-refinement constraint and used in the subsequent reachability check. So far, mutation detection has been realised by passing non-refinement constraints to the constraint solver one after the other for each action of the mutated action system. This has the advantage that only "real" semantical mutations are detected. However, for both use cases, we experienced that this is rather demanding in terms of runtime (cf. Section 6.6). A simpler and faster way to identify the mutated action is to perform mutation detection on a syntactic level, i.e., by comparing the source code of the actions. We are aware that this information could also be delivered from the mutation engine. However, we did not want to introduce unnecessary dependencies. Furthermore, it will be seen from our experiments that a syntactic check does not cause significant efforts (cf. Sections 7.2 and 7.3).

For the syntactic check, we have to consider the definitions of the actions. Additionally, their calls in the do-od block are important, where actual parameters could be manipulated, e.g., parameters could be replaced by constants or other variables. Our syntactic comparison is not sensitive to the renaming of

Algorithm 7.1 $\mathit{chkRef}(as, mutants) : unsafes$

```
1:  $unsafes := [\,]$
2:
3:  for all $asm \in mutants$ do
4:    $s := \mathit{getInitState}(as)$
5:    $visited := \{\}$
6:    $u := nil$
7:    while $s \neq nil$ do
8:      if $\mathit{unsafe}(s, as, asm)$ then
9:        $u := s$
10:       break
11:     end if
12:     $visited := visited \cup s$
13:     $s := \mathit{findNextState}(as, visited)$
14:   end while
15:   $unsafes.add(asm, u)$
16: end for
17: return $unsafes$
```

Algorithm 7.2 $\mathit{chkRef1}(as, mutants) : unsafes$

```
1:  $unsafes := [\,]$
2:  $states := \mathit{findAllStates}(as)$
3:  for all $asm \in mutants$ do
4:
5:
6:    $u := nil$
7:    for all $s \in states$ do
8:      if $\mathit{unsafe}(s, as, asm)$ then
9:        $u := s$
10:       break
11:     end if
12:   end for
13:
14:
15:   $unsafes.add(asm, u)$
16: end for
17: return $unsafes$
```

parameters and local variables, which are represented by Prolog variables. This is implemented via SICStus Prolog's term utilities library. The predicate `variant/2` checks whether two terms are identical modulo renaming of variables.

Our syntactic check requires some pre-conditions. Firstly, we do not support overloading of actions, i.e., each action is uniquely identified by its name. There must not exist two actions having the same name but a different number of parameters. Furthermore, we suppose that no action call is added/deleted from the do-od block by mutation operators. Note that actions may still be added/deleted from the transition relation by weakening/strengthening their guards. Finally, we do not allow the mutation of data types. If some of these pre-conditions are violated, we possibly miss a mutation. Our implementation checks the last assumption: each type defined in the original and in the mutated action system must have the same definition. The other assumptions are not checked automatically. Hence, we implemented our mechanism for finding the mutated action conservatively: If we cannot find any mutated action syntactically, we perform our semantic mutation detection strategy. However, using our mutation operators that respect our pre-conditions, this semantic mutation check is never triggered.

7.1.3 Pre-computation of Reachable States

In model-based mutation testing, we typically deal with a large set of mutated models derived from the original model. So far, the refinement checks between the original action system and each mutant were completely decoupled. However, the check for the reachability of unsafe states is always performed via an exploration of the same original action system (cf. Algorithm 6.2).

Algorithm 7.1 shows the straightforward approach for this reachability check for a given set of mutants. It is more abstract than Algorithm 6.2 as it focuses on the calculation of the reachable states. The inputs for Algorithm 7.1 are one original action system (as) and a set of corresponding mutated action systems ($mutants$). The result is a map $unsafes$ linking the mutants and their unsafe states. The algorithm iterates over the set of mutants (Line 3). The variable s represents the current state of the original action system, which is the initial state in the beginning (Line 4). Successor states are retrieved by the procedure *findNextState* (Line 13). It implements a breadth-first search, whereas it does not explore any state more than once (by considering a list of visited states). To ensure termination, it stops exploration at a user-specified depth limit. At each call, it returns the next reached state or nil in case of termination. Each state is tested whether it is an unsafe state (Line 8). If this is the case, the state space exploration is stopped and the mutant and the unsafe state are added into the map $unsafes$ (Line 15). If no unsafe state could be found, nil is inserted and the mutant is considered to refine the original up to the specified depth limit. For the next mutant, state space exploration is performed again. For the sake of simplicity, we omitted the recording of the traces to the unsafe states, the passing of the non-refinement constraints as parameters, etc.

An advantage of Algorithm 7.1 is that the state space is explored on demand, i.e., it is only explored until an unsafe state is found and not fully up to the given depth. For small sets of mutants, this is appropriate. For large sets of mutants, it is not very clever as the same state space is explored again and again. An alternative is to pre-compute all reachable states up to the given depth and then search for unsafe states in this set. Exploring the full state space up to the given depth is not really an overhead. It is done for equivalent mutants anyway and the probability that a large set of mutants contains at least one equivalent mutant is rather high.

Algorithm 7.2 describes the refinement check with a pre-computed state space. It takes the same input as Algorithm 7.1 and results in the same output. In contrast to Algorithm 7.1, Algorithm 7.2 explores the state space only once and then reuses the reached states during mutation analysis. The procedure *findAllStates* (Line 2) works analogously to *findNextState* of Algorithm 7.1, but does not return one reachable state after the other. Instead, it returns the full set of states that are reachable up to the given search depth at once. Afterwards, iteration over the mutants starts (Line 3), where each of the reached states is tested whether it is unsafe (Line 8). The loop in Line 7 iterates over the states with increasing depth to get the shortest counterexamples. Once an unsafe state is found, we stop searching for unsafe states (Line 10), save the result (Line 15), and proceed with the next mutant without exploring the state space again.

7.1.4 Incremental Solving

Incremental solving is a technique to efficiently solve a sequence of constraints $c_1, ..., c_n$ that have large parts in common. The constraints are related by adding or removing small parts. Incremental solving exploits the findings made during solving the constraint c_i for solving the subsequent constraint c_{i+1} [202]. Most modern SMT solvers, e.g., Microsoft's Z3 [76], MathSAT5 [67], or OpenSMT [57], provide this functionality. They regard their clause database, i.e., constraint store, as a stack where constraints can be pushed and popped. Pushing a constraint means that it is added by conjunction to the current constraints. The pop operation discards the most recently pushed constraint including clauses that were learnt due to this constraint. However, learnt clauses from constraints that are still on the stack remain unaffected.

Our refinement check is well suited to exploit incremental solving, as will be shown later. While most modern Satisfiability Modulo Theories (SMT) solvers support incremental solving, our used constraint solver does not offer such a functionality out of the box. Nevertheless, as we use Prolog as a programming language we were able to implement incremental solving using constraint logic programming and backtracking. Analogously to the incremental solving interface of SMT solvers, the constraint store is regarded to be a stack, where constraints can be pushed (posted) or popped (retracted). The method *solve* succeeds if the current store is satisfiable and a model may be retrieved. Otherwise, the constraints in the store are unsatisfiable and *solve* returns false. Pushing constraints is straightforward, while popping constraints has to be simulated via backtracking, i.e., we deliberately fail after we solved a constraint.

This principle is illustrated in Listing 7.1. We define a predicate *pushSolvePop*, which takes a constraint and returns a solution, i.e., a variable valuation that satisfies the constraint store. The first clause of this predicate, pushes the given constraint by calling it in Prolog (Line 2). Then it invokes the constraint

```
1  pushSolvePop(Constraint, _) :-
2      call(Constraint), % push given constraint on constraint store
3      solve(Solution),  % solve the current constraint store
4      assert(solution(Solution)), % assert solution to Prolog's database
5      fail.             % deliberately backtrack to remove the last
6                        % constraint from the constraint store
7
8  pushSolvePop(_, Solution) :-
9      retract(solution(Solution)). % retrieve and remove solution
10                                  % from Prolog's database
```

Listing 7.1: Prolog's backtracking facility is used for incremental solving.

solver to determine a solution for the current constraint store. This is encapsulated by a *solve* predicate that invokes the constraint solver's search for a solution (Line 3). Finally, the asserted constraint is removed from the constraint store by an explicit *fail* statement (Line 5). It causes Prolog to backtrack, which means that also the push operation of the constraint in Line 2 will be reverted. Furthermore, the found solution will be dropped. In order to preserve the solution, we *assert* it into Prolog's database as a fact with functor *solution*, before we fail (Line 4). In this way, we can retrieve it from the database in the alternative clause of our predicate in Lines 8 and 9. Note that the *retract* statement also removes the solution fact from Prolog's database. Overall, if the constraints in the store are satisfiable, our *pushSolvePop* predicate does not fail, but returns the desired solution. The predicate fails if the constraints are unsatisfiable. In this case, the first clause fails as the *solve* predicate fails if the constraint solver does not find a solution. Furthermore, the second clause cannot succeed, since the *retract* predicate fails as there is no solution fact in Prolog's database. In any case, the constraint store after calling the *pushSolvePop* predicate is in the same state as it was before calling the predicate. For background information on Prolog, we refer to Sterling and Shapiro's comprehensive book on Prolog [186].

Example 7.1. Assume that the current constraint store consists of the constraints X#>0 #/\ X#<10. Note that we directly use SICStus Prolog's syntax, where #> denotes *greater than*, #< means *smaller than*, and #/\ represents conjunction. Furthermore, #= denotes equality. By calling our predicate $pushSolvePop(X$#=2, $Solution)$, the given constraint will be pushed (Line 2 in Listing 7.1). The constraint store now comprises the constraints X#>0 #/\ X#<10 #/\ X#=2. In Line 3, these constraints are solved and the Prolog variable *Solution* will be unified with the only solution for these constraints, i.e., $X = 2$. Subsequently, this solution is asserted, i.e., the fact *solution(X = 2)* is added to Prolog's database. Finally, we fail, which causes Prolog to backtrack. The unification of the variable *Solution* with $X = 2$ is reverted and the constraint store is restored to its initial state, i.e., X#>0 #/\ X#<10. Backtracking causes Prolog to search for other solutions. Hence, the second clause of the *pushSolvePop* predicate is considered. There, the just asserted solution is retrieved and deleted from the store (Line 9) and the predicate succeeds. □

Note that push, solve, and pop can also be implemented as separate predicates by splitting Listing 7.1 into its individual parts. In the following, we will use these operations separately.

Algorithm 7.3 is a more detailed version of Algorithm 7.2 and gives additional information on the application of incremental solving. In Line 2, the original action system is translated. The resulting constraint system represents its transition relation ($trans\_rel$). It is posted to the constraint solver's store (Line 3). In Line 4, the state space is explored. The procedure *findAllStates* starts at the initial state of as and recursively searches for all possible successor states. It uses the *solve* method and can reuse the transition relation that is already in the constraint store. As the state space is now fully explored (up to a given depth limit), the transition relation is not needed any more and can be removed from the store

Algorithm 7.3 $\mathit{chkRef1Incremental}(as, \mathit{mutants}) : \mathit{unsafes}$

```
 1: $unsafes := [\,]$
 2: $trans\_rel := trans(as)$
 3: $solver.push(trans\_rel)$
 4: $states := \mathit{findAllStates}(as, solver)$
 5: $solver.pop()$   // pop trans_rel
 6: $solver.push(\neg trans\_rel)$
 7: for all $asm \in mutants$ do
 8:   $u := nil$
 9:   $a^m := \mathit{findMutatedAction}(as, asm)$
10:   $mut\_act := trans(a^m)$
11:   $solver.push(mut\_act)$
12:   for all $s \in states$ do
13:     $solver.push(v = s)$
14:     if $solver.solve()$ then
15:       $u := s$
16:       $solver.pop()$   // pop (v = s)
17:       break
18:     end if
19:     $solver.pop()$   // pop (v = s)
20:   end for
21:   $unsafes.add(asm, u)$
22:   $solver.pop()$   // pop mut_act
23: end for
24: $solver.pop()$   // pop $\neg$trans_rel
25: return $unsafes$
```

(Line 5). In exchange, the negated transition relation is required for each refinement check with a mutant (cf. Theorem 6.1). It is added to the store in Line 6. The actual refinement check starts in Line 7. It iterates over the set of mutants. In Line 9, *findMutatedAction* syntactically compares the original and the mutated action system. Thereby, it identifies the mutated action a^m, which represents the second part of our non-refinement constraint (cf. Section 6.3). It is translated into constraints (Line 10) and added to the solver's store (Line 11), which now contains the complete non-refinement constraint for the current mutant. The loop in the Lines 12 to 20 performs the search for an unsafe state in the list of reachable states. Each state s is used as the pre-state v of the non-refinement constraint (Line 13). If the current constraint store is satisfiable, we just found an unsafe state – a state from which the mutant behaves in a way that is not specified by the original (Lines 14 and 15). In this case we stop iterating over the states (Line 17). In any case, the constraint $v = s$ is removed from the store (Line 16 and Line 19 respectively). To process the next mutant, the part of the non-refinement constraint that is specific to the current mutant has to be removed from the store (Line 22). After finishing all mutants, we finally remove the negated transition relation from the constraint store (Line 24).

Algorithm 7.3 shows that both the reachability analysis and the check for unsafe states are well suited to exploit incremental solving. During reachability, the transition relation is solved again and again – only the pre-states change (Line 4). While testing states whether they are unsafe, the non-refinement constraint has to be solved repeatedly – again with changing pre-states (Line 13). Each non-refinement constraint contains the negated original (Line 6). Thus, when processing several mutants, there is a common part remaining in the store.

Algorithm 7.4 shows the incremental version of Algorithm 7.1. The use of incremental solving is very similar to Algorithm 7.3. However, note that for Algorithm 7.4, we use two instances of the solver. One for solving the transition relation ($solver\_tr$), the other one for solving the non-refinement constraint ($solver$). As the exploration of the state space and the non-refinement checks interleave, both the transition relation and the negated transition relation are needed in the constraint store at the same time. Using only one solver would result in an inconsistent constraint store.

7.1.5 Analysis of Optimisations

Roughly speaking, all of our refinement checking approaches with/without optimisations are in the same class of complexity. As we check for the satisfiability of constraint systems, we are dealing with NP-complete problems.

Algorithm 7.4 $chkRefIncremental(as, mutants) : unsafes$

```
1: $trans\_rel := trans(as)$
2: $solver\_tr.push(trans\_rel)$
3: $solver.push(\neg trans\_rel)$
4: $unsafes := [\,]$
5: **for all** $asm \in mutants$ **do**
6:     $s := getInitState(as)$
7:     $visited := \{\}$
8:     $u := nil$
9:     $a^m := findMutatedAction(as, asm)$
10:    $mut\_act := trans(a^m)$
11:    $solver.push(mut\_act)$
12:    **while** $s \neq nil$ **do**
13:        $solver.push(v = s)$
14:        **if** $solver.solve()$ **then**
15:            $u := s$
16:            $solver.pop()$   // pop (v = s)
17:            **break**
18:        **end if**
19:        $solver.pop()$   // pop (v = s)
20:        $visited := visited \cup s$
21:        $s := findNextState(as, visited, solver\_tr)$
22:    **end while**
23:    $unsafes.add(asm, u)$
24:    $solver.pop()$   // pop mut_act
25: **end for**
26: $solver\_tr.pop()$   // pop trans_rel
27: $solver.pop()$   // pop ¬trans_rel
28: **return** $unsafes$
```

Nevertheless, we analyse our refinement checking approaches for their potential of improving efficiency. A practicable method is to give the upper bound for the number of constraint solver calls required. We are aware that there is not necessarily a direct correlation to solving time. However, this metric still gives valuable insights. The number of solver calls of our algorithms depends on the number of mutated models ($|mutants|$), the number of actions defined in the action systems ($|actions|$), the number of states ($|states|$), and the number of transitions in the equivalent Labelled Transition System (LTS) (cf. Definition 3.12) ($|transitions|$). It holds that $|transitions| \leq |states|^2 \times |actions| \times |parameter\ valuations|$. For our unoptimised refinement checker as described in Section 6, we call the solver at most once per action for finding the mutated action. For the state space exploration, the solver is called at most $|transitions| + |states|$ times: $|transitions|$ times for enumerating the enabled transitions, and $|states|$ times to find out that there are no further transitions enabled in the states. This is an upper bound as the exploration is stopped as soon as an unsafe or visited state is found. In this search for unsafe states, the solver is called at most once per state. Hence, the upper bound for the number of solver calls in the unoptimised algorithm for all mutants is

$$|mutants| \times (\underbrace{|actions|}_{find\ mutated\ action} + \underbrace{|transitions| + |states|}_{state\ space\ exploration} + \underbrace{|states|}_{check\ for\ unsafe\ states})$$

This limit also holds for our first optimised version (Section 7.1.1). The different variable and value selection heuristics are implemented in the constraint solver itself. Hence, they do not influence the number of solver calls. As we deal with heuristics, the performance strongly depends on the given problems and no general best solution can be predicted. Nevertheless, the variable selection heuristics *first-fail principle* and *most-constrained heuristic* are more sophisticated. They dynamically adapt to the current state of the search. Thereby, they have better chances of achieving good results.

Syntactic mutation detection (Section 7.1.2) reduces the number of solver calls. For finding the mutated action, no solver calls are required any more. Hence, the upper bound for the number of solver calls for all mutants is reduced to

$$|mutants| \times (\underbrace{|transitions| + |states|}_{state\ space\ exploration} + \underbrace{|states|}_{check\ for\ unsafe\ states})$$

The upper bound for the number of solver calls for all mutants is furthermore reduced by the precomputation of the state space (Section 7.1.3). As the state space is only explored once (not for every mutant), it reduces to

$$\underbrace{|transitions| + |states|}_{state\ space\ exploration} \quad + \quad (|mutants| \quad \times \quad \underbrace{|states|}_{check\ for\ unsafe\ states} \quad)$$

This also holds if incremental solving is applied (Section 7.1.4). With incremental solving, parts of the constraint systems can be reused. More specifically, the transition relation of the original action system can be reused at most $|transitions|+|states|$ times. Its negation (for the non-refinement constraints) can be reused for each mutant, i.e., at most $|mutants| \times |states|$ times. Finally, we can reuse the constraints for the mutated actions. However, this varies for each mutant. Hence, we can reuse $|mutants|$ constraints – each one for at most $|states|$ times.

To assess our optimised implementations, we repeated our previous experiments described in Section 6.6. We start by reporting results for the CAS.

7.2 Experiments with the Car Alarm System

For our experiments with the CAS, we use the same setting as in our previous experiments described in Section 6.6.1. We deal with the same four versions of our CAS model: CAS_1 with parameter values 20, 30, and 270 for waiting, CAS_10 with parameter values multiplied by 10, CAS_100 with parameters multiplied by 100, and CAS_1000 with parameters multiplied by 1000. For each CAS version, we reuse our 207 mutated models. Furthermore, our experiments were conducted on the same computer, i.e., on a MacBook Pro with an Intel i7 dual-core processor (2.8 GHz) and 8 GB RAM with a 64-bit operating system. Hence, the results of our optimised implementation can be directly compared with our previous results.

7.2.1 Variable and Value Selection Heuristics

We experimented with the constraint solver's search strategy by trying different combinations of variable and value selection strategies (see Section 7.1.1). Variable selection strategies include: (1) *leftmost*, which selects variables from left to right in a given list, (2) *ff* - the first-fail principle, and (3) *ffc* - the most-constrained heuristic. For value selection, *up* or *down* may be chosen, i.e., values are either selected in ascending or in descending order.

We tried all six combinations of these variable and value selection strategies. The runtimes of our refinement checker using the default setting (*leftmost-up*) were already reported in Table 6.1. However, we restate them in Table 7.1 to have an overview of all strategies. Again, we partition the total runtime into the time needed for finding the mutated action "*1*" and the time for the combined reachability and non-refinement check "2". Note that the semantic mutation detection via the constraint solver is used for finding the mutated action. Results with syntactic mutation detection will be reported in the next section.

The default setting *leftmost-up* is the worst setting for our example. It takes up to 3.4 hours to deal with one mutant of the CAS_1000 model (see Table 6.1). As can be seen from Table 7.1, also *leftmost-down* does not scale well for larger parameter domains. For CAS_100 and CAS_1000, some mutants take particularly long. For example, for CAS_1000 the maximum time spent on one mutant is 509 seconds, which is the main part of the time needed for all mutants. 75% of all mutants do not take longer than 0.17 seconds per mutant. While *leftmost-up* also showed outliers, the other four combinations did not. Note that information about outliers is given in Table B.2 in the appendix by stating values for quartiles. In general, the first-fail principle (*ff*) as well as the most-constrained heuristic (*ffc*) show good results regardless of the value selection strategy and the size of the variable domains. This makes sense as these heuristics are more sophisticated than the static selection of the *leftmost* variable. They adapt to

		CAS_1			CAS_10			CAS_100			CAS_1000		
		Σ	ϕ	max	Σ	ϕ	max	Σ	ϕ	max	Σ	ϕ	max
leftm.-up	1	23	0.11	13	160	0.77	127	32.4 min	9.39	28 min	4.2 h	73	3.4 h
	2	18	0.09	0.37	19	0.09	0.38	23	0.11	0.49	18	0.09	0.35
	total	**41**	0.2	13	**179**	0.86	127	**33 min**	9.5	28 min	**4.2 h**	73	3.4 h
leftm.-down	1	9	0.04	0.55	13	0.07	5.25	87	0.42	75	517	2.5	509
	2	18	0.09	0.35	18	0.09	0.34	24	0.12	0.47	17	0.08	0.34
	total	**27**	0.13	0.63	**31**	0.16	5.33	**111**	0.54	75	**534**	2.58	509
ff-up	1	31	0.15	0.31	32	0.15	0.31	45	0.22	0.43	31	0.15	0.3
	2	18	0.09	0.46	18	0.09	0.45	25	0.12	0.75	18	0.09	0.44
	total	**49**	0.24	0.68	**50**	0.24	0.68	**70**	0.34	1.1	**49**	0.24	0.67
ff-down	1	38	0.18	0.35	38	0.18	0.37	54	0.26	0.53	37	0.18	0.36
	2	19	0.09	0.38	19	0.09	0.37	26	0.13	0.51	19	0.09	0.35
	total	**57**	0.27	0.56	**57**	0.27	0.61	**80**	0.39	0.89	**56**	0.27	0.56
ffc-up	1	24	0.12	0.32	23	0.11	0.26	32	0.16	0.33	22	0.11	0.26
	2	19	0.09	0.49	19	0.09	0.47	26	0.12	0.62	18	0.09	0.46
	total	**43**	0.21	0.66	**42**	0.2	0.65	**58**	0.28	0.84	**40**	0.2	0.62
ffc-down	1	25	0.12	0.26	26	0.12	0.27	35	0.17	0.37	24	0.12	0.26
	2	20	0.09	0.38	20	0.1	0.56	27	0.13	0.6	18	0.09	0.35
	total	**45**	0.21	0.48	**46**	0.22	0.58	**62**	0.3	0.76	**42**	0.21	0.46

Table 7.1: Execution times of our refinement checker using different variable/value selection strategies for the four CAS versions. *"1"* stands for "find mutated action", *"2"* for "reach & non-refine", and *"total"* is the sum thereof. All values are given in seconds unless otherwise noted.

the current state of the search and select the next variable dynamically. Nevertheless, the combination *leftmost-down* also accomplishes good results. For example, it achieves the shortest runtimes (27 and 31 seconds respectively) for CAS_1 and CAS_10. For CAS_100 and CAS_1000, *ffc-up*, i.e., the combination of the most-constrained heuristic and the ascending order for value selection, is the fastest combination. So there is no setting that consistently performs best for all CAS versions. Nevertheless, using the first-fail principle or the most-constrained heuristic seems to be the best overall choice. They scale for all four CAS versions (although CAS_100 seems to be a small outlier). We will check this hypothesis in our second experiment with the particle counter use case (Section 7.3).

7.2.2 Mutation Detection Strategies

Although the above results are already promising, they can be further improved. As can be seen in Table 7.1, the time needed for finding the mutated action "*1*" still takes a considerable amount of time (33 - 97% of the total time needed for refinement checking). As already proposed in Section 7.1.2, a syntactic analysis to find the mutated action should solve this problem. Table 7.2 lists the execution times in seconds needed for syntactic mutation detection "*1*" for our four CAS versions and the six combinations of variable and value selection strategies. Syntactic mutation detection leads to runtimes that are drastically decreased compared to semantic mutation detection using the constraint solver. For each CAS version and for each combination of variable/value selection strategies, the time to find the mutated action for all 207 mutants is below one second. Hence, the total time needed for refinement

		CAS_1			CAS_10			CAS_100			CAS_1000		
		Σ	ϕ	max	Σ	ϕ	max	Σ	ϕ	max	Σ	ϕ	max
leftm.-up	1	0.54	0	0.08	0.57	0	0.07	0.48	0	0.07	0.51	0	0.08
	total	**20**	0.1	1.25	**20**	0.1	1.23	**20**	0.1	1.24	**20**	0.1	1.22
leftm.-down	1	0.57	0	0.08	0.52	0	0.08	0.54	0	0.09	0.6	0	0.08
	total	**19**	0.09	0.39	**20**	0.1	0.4	**20**	0.1	0.47	**20**	0.09	0.4
ff-up	1	0.7	0	0.26	0.69	0	0.24	0.85	0	0.33	0.86	0	0.26
	total	**19**	0.09	0.48	**19**	0.09	0.47	**23**	0.11	0.49	**23**	0.11	0.59
ff-down	1	0.8	0	0.31	0.81	0	0.28	0.9	0	0.31	0.96	0	0.29
	total	**20**	0.09	0.38	**19**	0.09	0.37	**23**	0.11	0.45	**22**	0.11	0.45
ffc-up	1	0.69	0	0.19	0.56	0	0.18	0.7	0	0.21	0.65	0	0.21
	total	**19**	0.09	0.48	**19**	0.09	0.46	**23**	0.11	0.49	**20**	0.1	0.47
ffc-down	1	0.71	0	0.21	0.55	0	0.2	0.63	0	0.21	0.56	0	0.19
	total	**20**	0.1	0.37	**19**	0.09	0.36	**21**	0.1	0.37	**19**	0.09	0.37

Table 7.2: Execution times for the CAS case study with our refinement checker using syntactic analysis for finding the mutated action (*"1"*). Time values for "reach & non-refine" are missing as they are almost the same as in Table 7.2. All values are given in seconds.

checking now basically consists of the time needed for the combined reachability and non-refinement check. For this Step "*2*" of our process, we have omitted the runtimes in Table 7.2 as they are almost the same as in Table 7.1. Using syntactic mutation detection, we achieved runtimes of 19 - 23 seconds to process all 207 mutated models for each version of the CAS. Hence, the settings for the constraint solver on how to choose variables and values have become irrelevant. The appendix includes Table B.3, which extends Table 7.2 by values for the quartiles.

7.2.3 Pre-computation of Reachable States

All results presented so far were based on Algorithm 7.1, where the state space of the original action system was explored for each mutant. In Section 7.1.3, we proposed a pre-computation of all reachable states up to a certain depth (Algorithm 7.2).

Table 7.3 gives an overview of the computation times for all mutants of each CAS version using the combination of the most-constrained heuristic with ascending value selection. Row "*Algorithm 7.1*" restates the execution times needed without the pre-computation of reachable states with syntactic mutation analysis (cf. Table 7.2). Additionally, Table 7.3 gives values for Algorithm 7.2, which performs the reachability analysis only once. Here, we divide the total runtime into (1) the time needed for the state space exploration ("*reach*"), which is performed only once for all mutants, and (2) the time needed to find an unsafe state in the set of pre-computed states ("*find unsafe*"). The total runtime is a bit higher than the sum of these two items as it also contains input/output operations such as parsing or logging of the results. For each CAS version, the time needed for refinement checking 207 mutants could be further decreased from about 20 seconds to approximately 7 seconds by the pre-computation of the state space.

7.2.4 Incremental Solving

Our last suggested improvement concerned the use of incremental solving as explained in Section 7.1.4. We implemented incremental versions of Algorithm 7.1 as well as of Algorithm 7.2 resulting in Algorithm 7.4 and Algorithm 7.3. The execution times of these two algorithms on the CAS case study are

		CAS_1	CAS_10	CAS_100	CAS_1000
Algorithm 7.1	total	**19**	**19**	**23**	**20**
	reach	0.07	0.08	0.07	0.08
Algorithm 7.2	find unsafe	5.64	5.28	5.75	5.08
	total	**7.03**	**6.6**	**7.02**	**6.38**
Algorithm 7.4	total	**2.82**	**2.84**	**3.38**	**8.55**
	reach	0.01	0.01	0.02	0.01
Algorithm 7.3	find unsafe	1.5	1.59	1.53	1.71
	total	**2.63**	**2.62**	**2.66**	**2.69**

Table 7.3: Execution times for the CAS case study. Different versions of our refinement checker with/without optimisations regarding the state space pre-computation and incremental solving have been run using the most-constrained heuristic combined with ascending value selection (*ffc-up*). All values are given in seconds.

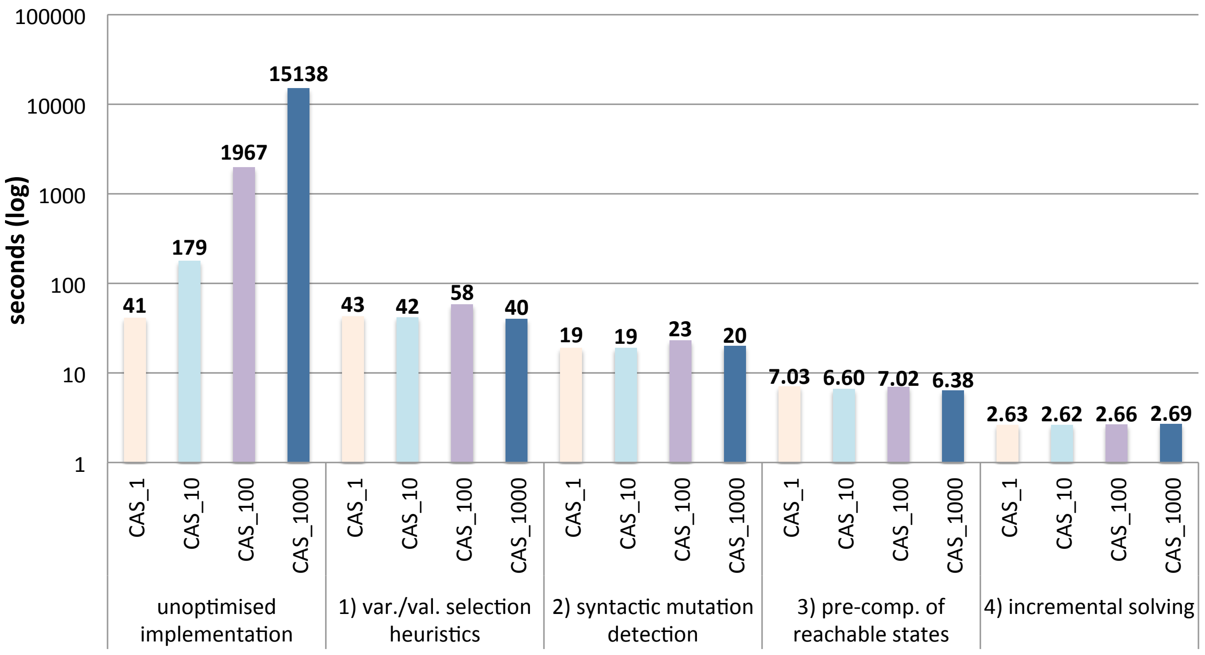


Figure 7.1: Reduction of the computation time for the CAS case study.

given in Table 7.3. Algorithm 7.4 achieves runtimes from 2.82 to 8.55 seconds for checking refinement of 207 mutants. The runtimes are not constant with increasing domains of the parameters. Algorithm 7.3 performs better. It is faster (less than 3 seconds per CAS version) and the runtime is constant. Again, the options *ffc-up* were used for constraint solving as our experiments in Section 7.2.1 indicate that this combination is a reasonable choice.

To summarise the results of our optimisations, Figure 7.1 compares the computation times of our implementations for the CAS case study. The stated values give the time required to process all 207 mutated models. As can be seen from the diagram, each optimisation was beneficial. Note that the time in seconds on the Y-axis is scaled logarithmically, i.e., each grid line means an improvement by factor 10. The left-hand side of the diagram shows the execution time required by our basic implementation as described in Chapter 6, which did not scale for larger parameter domains. By applying more sophisticated variable and value selection heuristics of the constraint solver, the runtimes could be drastically reduced and stabilised for all CAS versions. For the combination of the most-constrained heuristic for variable

Algorithm 7.1		semantic		syntactic		
		Σ	mutants	Σ	ϕ	max
leftmost-up	1	> 5.98 h		23.8	0.04	3.6
	2	> 41		2.2 h	11.5	52.9
	total	$>$ **6 h**	5/672	**2.2 h**	11.6	53.0
leftmost-down	1	> 5.99 h		18.8	0.03	2.9
	2	> 1		1.8 h	9.6	44.1
	total	$>$ **6 h**	4/672	**1.8 h**	9.6	44.1
ff-up	1	> 5.98 h		17.0	0.03	0.6
	2	> 87		1.9 h	10.2	39.4
	total	$>$ **6 h**	7/672	**1.9 h**	10.2	39.4
ff-down	1	> 5.96 h		16.0	0.02	0.29
	2	> 2.4 min		2.2 h	11.9	47.2
	total	$>$ **6 h**	8/672	**2.2 h**	11.9	47.2
ffc-up	1	> 5.97 h		16.9	0.03	0.6
	2	> 1.4 min		1.9 h	10.2	39.4
	total	$>$ **6 h**	6/672	**1.9 h**	10.2	39.4
ffc-down	1	> 5.97 h		15.8	0.02	0.3
	2	> 1.6 min		2.2 h	11.8	46.9
	total	$>$ **6 h**	7/672	**2.2 h**	11.8	46.9

Table 7.4: Execution times of our refinement checker for the particle counter based on Algorithm 7.1 using semantic and syntactic mutation detection strategies. All six variable/-value selection strategies were used. *"1"* stands for "find mutated action", *"2"* for "reach & non-refine", and *"total"* is the sum thereof. All values are given in seconds unless otherwise noted.

selection and the ascending value selection, we achieved computation times between 40 and 58 seconds. By our syntactic mutation detection, the runtimes could be further decreased by approximately 50% and now vary between 19 and 23 seconds. The pre-computation of the reachable states could reduce the runtimes to approximately 7 seconds for all mutants for each CAS version. Finally, incremental solving decreased the computation time once more and achieves runtimes of less than 3 seconds.

In the next section, we investigate whether our optimisations show equally good results for the particle counter use case.

7.3 Experiments with the Particle Counter

Like for the CAS, we ran our optimised refinement checker implementations on the particle counter use case. Again, we used the same setting as in Section 6.6.2, i.e., the same action system model, the same 672 mutated models, and the same computer to run our experiments.

7.3.1 Variable and Value Selection Heuristics and Mutation Detection Strategies

Table 7.4 summarises results for our first two optimisations applied on the particle counter use case. It contains values for the runtimes of two different implementations of our refinement checker: one uses semantic mutation detection, the second uses syntactic mutation detection (cf. Section 7.1.2). Both use Algorithm 7.1, i.e., they explore the state space several times (once for each mutant). Both implementations were run six times due to varying configurations for the constraint solver. For each configuration,

		ffc-up			*leftmost-down*		
		Σ	ϕ	max	Σ	ϕ	max
Algorithm 7.1	total	**1.9 h**	10.2	39.4	**1.8 h**	9.6	44.1
	reach	51.4	0.07	-	52.7	0.08	-
Algorithm 7.2	find unsafe	44.1 min	3.9	17.0	42.8 min	3.8	17.4
	total	**45.1 min**	4.0	-	**43.8 min**	4.0	-
Algorithm 7.4	total	**32.1 min**	2.9	18.3	**33.2 min**	3.0	17.3
	reach	22.4	0.03	-	27.01	0.04	-
Algorithm 7.3	find unsafe	1.6 min	0.2	2.23	1.7 min	0.2	1.5
	total	**2.0 min**	0.2	-	**2.2 min**	0.2	-

Table 7.5: Execution times for the particle counter. Different versions of our refinement checker with/without optimisations regarding the state space pre-computation and incremental solving have been run in two constraint solver configurations: *ffc-up* and *leftmost-down*. All values are given in seconds unless otherwise noted.

we divide the total computation time *"total"* into two parts: *"1"* stands for the time needed to find the mutated action, and *"2"* represents the time needed for the combined reachability and non-refinement check. Note that the semantic mutation detection with the default constraint solver settings *leftmost-up* corresponds to the results of our unoptimised implementation already reported in Section 6.6.2.

The implementation using semantic mutation detection performed rather poor. For none of the six constraint solver strategies, it managed to check all 672 mutants for refinement with the original model in a reasonable amount of time. We quit the execution of each run after a timeout of 6 hours. The progress can be characterised in terms of mutants being processed. For no configuration, more than 8 mutants could be handled within 6 hours. As in the CAS case study (cf. Table 7.1), the vast majority of the overall computation time is needed for finding the mutated action. Again, syntactic mutation analysis resolves this problem. The time required for finding the mutated action syntactically lies between 16 and 24 seconds. We would expect that this time is constant for all constraint-solver configurations. We repeated the experiments and found out that this variance is reproducible. We assume that it is caused by internals of the Prolog runtime, e.g., by different needs for garbage collection. Anyway, these times required for finding the mutated actions does not have great influence on the overall computation times any more. The total time for the refinement check is 1.8 to 2.2 hours for checking all 672 mutants. Hence, each constraint solver configuration achieves almost equally good results. The average time per mutant amounts to $\sim$10 to $\sim$12 seconds. Again, there are a couple of mutants taking longer than most others. The median value for checking one mutant is around 2 seconds. 75% of the mutants can be processed in $\leq$ 14 to 20 seconds per mutant. The time needed for finding the mutated actions shrank from several hours to a few seconds and takes almost equally long for each mutant. As in the CAS experiment, the strategy *ffc-up* still belongs to the fastest strategies. Note that the appendix includes Table B.4, which extends Table 7.4 with values for quartiles.

Due to the syntactic mutation analysis, all 672 model mutants could be processed. Thereof, 121 refine the original model. The remaining 551 mutants have unsafe states between depth 0, i.e., the initial state is unsafe, and depth 21.

7.3.2 Pre-computation of Reachable States and Incremental Solving

Table 7.5 presents the effect of our last two optimisations on the particle counter model. We ran all experiments with two constraint solver configurations: *ffc-up*, which performed very good for the CAS (cf. Table 7.1), and *leftmost-down*, which achieved the best results for the particle counter with syntactic

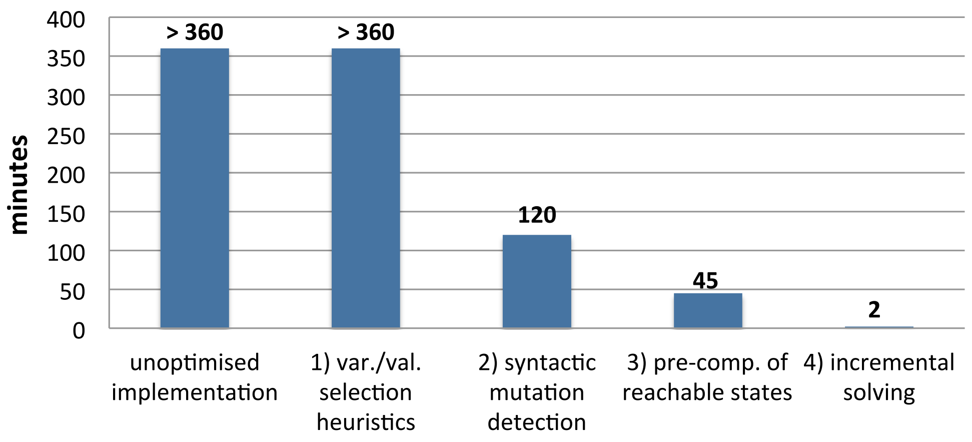


Figure 7.2: Reduction of the computation time for the particle counter use case.

mutation detection (cf. Table 7.4). For both strategies, we state the time required for all mutants (Σ), the average time for one mutant (ϕ), and the maximum time required by one mutant (max). The first data row (Algorithm 7.1) restates the execution times from Table 7.4 for the two constraint solver configurations. Algorithm 7.2 improves Algorithm 7.1 by exploring the state space of the original action system only once (cf. Section 7.1.3). For Algorithm 7.2, the table distinguishes between various sub-tasks. The row "*reach*" states the time needed for the exploration of the state space up to the maximum depth of 28. Note that we do not state a maximum value per mutant for "*reach*", as the state space is explored only once for all mutants. The row "*find unsafe*" gives the time required to check these states for non-refinement. The row "*total*" states the overall time needed for the refinement check. For Algorithm 7.1, we cannot make this distinction since these tasks are entangled. The computation of the state space takes almost 1 minute for both solver strategies. For 672 mutants, it is a considerable performance reduction to do it just once. The runtimes could be reduced from almost 2 hours to about 45 minutes, a reduction of $\sim$58%.

Table 7.5 further contains our results regarding the exploitation of incremental solving (cf. Section 7.1.4). Algorithm 7.4, which is the incremental version of Algorithm 7.1, achieves runtimes of 32 and 33 minutes respectively. It is faster than Algorithm 7.2. Hence, incremental solving was able to reduce the runtime from almost two hours to half an hour for the algorithm re-exploring the state space for each mutant. This is a reduction of almost 75%. For Algorithm 7.2, which explores the state space only once, incremental solving reduced the runtime from 45 to 2 minutes (Algorithm 7.3). Here, the performance gain is even higher: a reduction by 95%. For the incremental algorithms, the pre-computation of the state space reduces the runtime from 32 to 2 minutes, meaning a reduction by $\sim$94%. Each setting produced some outliers. For example, Algorithm 7.1 using strategy *ffc-up* has an arithmetic mean of 10.2 seconds, whereas the median value is less than 2 seconds. 75% of the mutants could be processed in $\leq$ 17 seconds per mutant. Further details can be found in Table B.5 in the appendix. It extends Table 7.5 by giving values for the quartiles.

Figure 7.2 shows the reduction of the computation times for the particle counter achieved with our optimisations. The stated values give the time required to process all 672 mutated models. The left-hand side of the diagram shows the execution time required by our basic implementation as described in Chapter 6. While the application of more sophisticated variable and value selection heuristics of the constraint solver helped with our CAS models (cf. Figure 7.1), the runtimes did not improve noticeably for the particle counter. In both cases, we stopped our experiments after a timeout of 6 hours as no significant progress could be observed. Only by our syntactic mutation detection, the runtimes could be significantly decreased to approximately 2 hours. The pre-computation of the reachable states could further reduce the runtimes to approximately 45 minutes. Finally, incremental solving achieved a further performance gain. The 672 mutants can be checked for refinement in approximately 2 minutes.

As can be seen from Figure 7.1 and Figure 7.2, our optimisations significantly reduced the computation times for the simple CAS as well as for the more complex, industrial particle counter use case and led to a highly-optimised implementation of a refinement checker for action systems.

8 Test Case Construction

Parts of this chapter are going to be published in an issue of the SCP journal [17].

The previous two chapters concentrated on the refinement check between a mutated and an original action system. In case of non-refinement, it results in an unsafe state and a trace leading to this state. However, model-based mutation testing aims at generating test cases. In this chapter, we explain how we construct test cases from traces leading to unsafe states and report results from experiments with our two use cases. As in the previous two chapters, we concentrate on plain action systems as defined in Section 5.2.2. The required extensions to support complex action systems will be discussed in Chapter 9.

8.1 Test Case Construction Approach

If the mutated action system does not refine the original, our refinement check provides an unsafe state (Definition 6.1) and a trace leading to this state, which we extend to a test case.

Remember that our action system language (cf. Section 5.2) provides means to classify actions as input or output actions. So far, we did not use this classification as refinement does not distinguish between inputs and outputs. However, for testing, a well-defined testing interface (Definition 2.9) is essential. In case of an input action in the test case, the tester has to become active and has to send the input to the SUT. In case of outputs, the tester has to be ready to receive outputs from the SUT and checks whether they are specified in the test case. Remember that inputs to the SUT are also denoted as *controllable*, because they are controlled by the tester. Similarly, outputs from the SUT are also called *observable*, since they can be observed by the tester. In our test cases, we prefix controllable actions by *ctr* and observable actions by *obs* to comply with MoMuT::UML's existing test case generation backend Ulysses (cf. Section 1.5.1). A further guideline of MoMuT::UML is the generation of test cases that satisfy the following properties defined by the ioco theory, which we introduced in Section 3.2.

Definition 8.1 (ioco Test Case)
According to the ioco theory [191], a test case is a Labelled Transition System (LTS) with inputs and outputs $\langle S, (L \cup \{\delta\}), T, s_0 \rangle$ (cf. Definition 3.12) with the following properties:

1. It is deterministic (cf. Definition 3.15).
2. It has finite behaviour (cf. Definition 3.14).
3. All of its terminal states assign a verdict (cf. Definition 2.19).
4. The test case is controllable, i.e., each state either specifies one input to be sent to the SUT or accepts all outputs including quiescence (Definition 3.16) from the SUT.

The last property concerning controllability may be changed by requiring that a test case always accepts any output of the SUT [171, 192]. As a consequence, during test execution it has to be decided non-deterministically whether to send an input or to wait for an output from the SUT. In order to conform to MoMuT::UML's existing test case generator Ulysses, we stick to the original definition.

A further property of the generated test cases, which was predefined by MoMuT::UML, is that the produced test cases are positive test cases (Definition 2.14). We do not explicitly state fail verdicts. Hence, we satisfy the property that a test case accepts all outputs from the SUT (Definition 8.1, Item 4) only implicitly. We presume that every observable action not specified by the test case leads to a fail verdict. Thus, the test case is a subset of the behaviour of the original model. Note that the original action system is sufficient for expanding a given trace to a test case. The pass verdict is characterised by successfully passing an unsafe state, i.e., only specified observable actions are allowed after an unsafe state. Hence, we extend the trace to the unsafe state by all observable actions that are enabled in the unsafe state in the original model. Each of these transitions leads to a pass verdict.

For non-deterministic systems (Definition 2.12), we additionally need inconclusive verdicts. Reaching an inconclusive verdict does not mean that the test case failed, but that the test purpose, which is a given unsafe state in our case, could not be reached in the test run. Consider a non-deterministic SUT, which is allowed to choose between several possible output actions. If only one of these outputs leads to the test purpose, but the SUT chooses another one, the SUT behaves correctly but cannot reach the test purpose any more. We augment our test cases with inconclusive verdicts by following the trace to the unsafe state. In each state followed by an observable action, we test whether alternative observable actions are enabled. If this is the case, we add a transition leading to an inconclusive verdict for each additional observable action. Note that we only add transitions to states followed by observable actions, but not to states that specify controllable actions, in order to respect the controllability property of ioco test cases (Definition 8.1, Item 4).

To comply to the ioco theory [191], we have to consider quiescence (cf. Definition 3.16), which is disregarded during our refinement check. Quiescence is represented by an additional observable action δ. As we concentrate on plain action systems for now, we do not have internal events and thus, the δ-action is only enabled whenever there are no other observable actions. As a consequence, δ may only occur directly before pass verdicts in our refinement-based test cases. Earlier in the test case, it cannot be enabled whenever there are other observable actions. If there is a controllable action in the test case, there must not be observable actions (cf. Definition 8.1, Item 4). Furthermore, it is not part of the trace to the unsafe state itself as our refinement relation does not consider quiescence.

To implement the above described test case construction based on LTSs, we explore the relevant parts of the action system's state space. As in the reachability analysis for our refinement check, the transition relation is encoded as a constraint satisfaction problem using our predicative relational semantics of action systems (cf. Section 6.3.2). The exploration is started at the initial state of the action system and the enabled actions and their successor states are determined by repeatedly solving the transition relation. This yields the LTS semantics of the action system, where the actions form the alphabet of the LTS and the states of the action system correspond to states of the LTS. For the use with ioco, the resulting LTS still has to be enriched by δ-transitions to incorporate quiescence, which is not considered by the constraints representing the transition relation. By Definition 8.1 (Item 4), an ioco test case is a deterministic LTS. Hence, if the obtained LTS is not deterministic, i.e., if any trace of the LTS leads to more than one state, it is determinised to generate a deterministic test case. A standard approach for determinisation is the subset construction [120].

Example 8.1. The LTS representing the behaviour of the action system model of the CAS, which was presented in Section 5.2.1, is depicted in Figure 8.1. Like in Section 3.2, the LTS is represented as a graph. Remember that controllable actions are marked by prefix *ctr*, while observable actions have the prefix *obs*. Furthermore, the first parameter of each action denotes time (cf. Listing 5.1). For controllable actions, it states the number of time units the tester has to wait before sending the input to the SUT. For observable actions, it denotes the number of time units after which the SUT might deliver an output. Note that this LTS is already deterministic (Definition 3.15). □

In our implementation, we do not fully explore the action system and compute the complete underlying LTS to extend one specific trace into a test case. In practice, we step through the given trace and only explore those parts of the action system that are relevant to the test case. If we assign a verdict to a state in the test case, we do not need to further explore this state. Also determinisation is performed on the fly. If an action leads to more than one successor state, we merge all of them into one LTS state. If

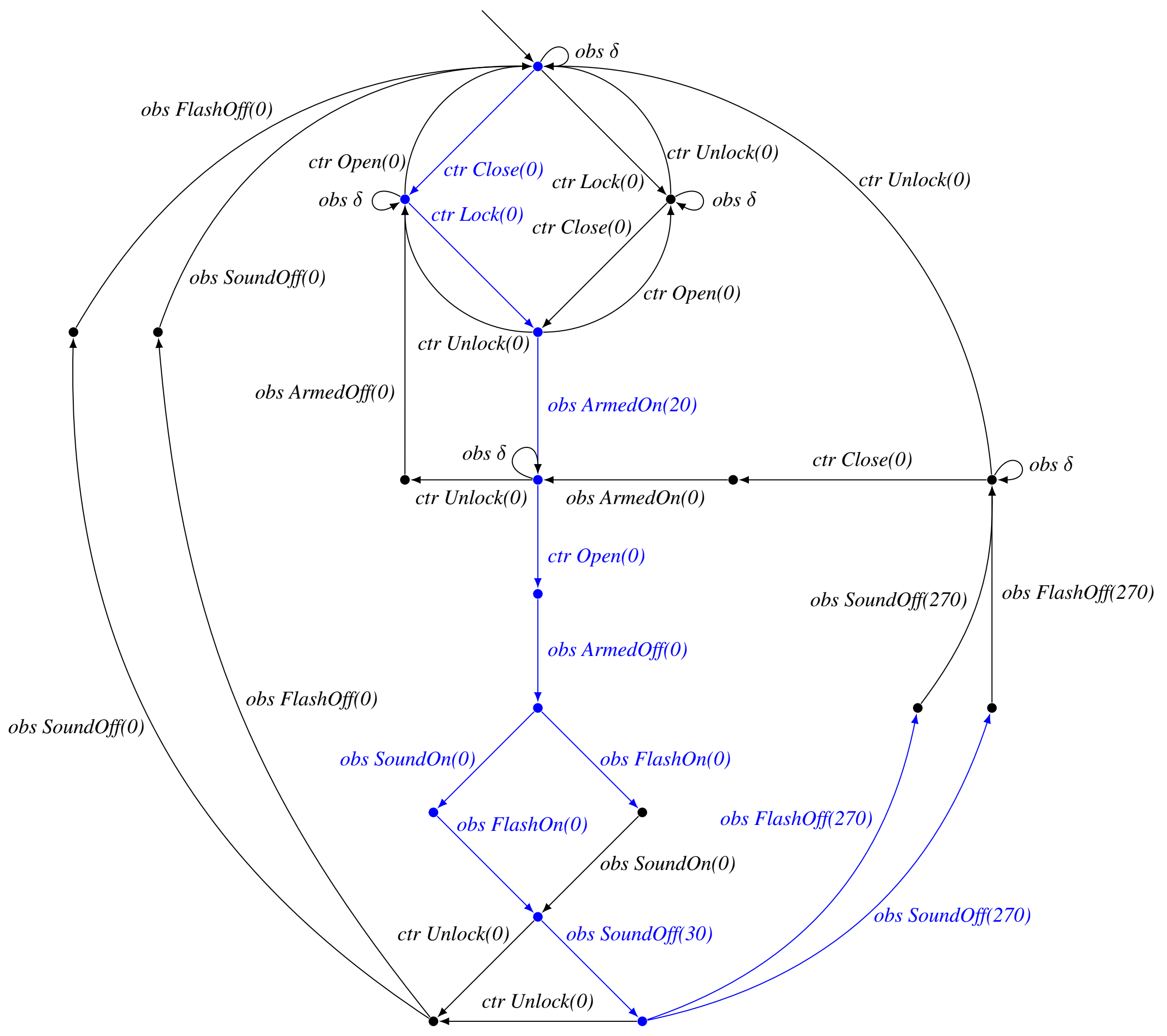


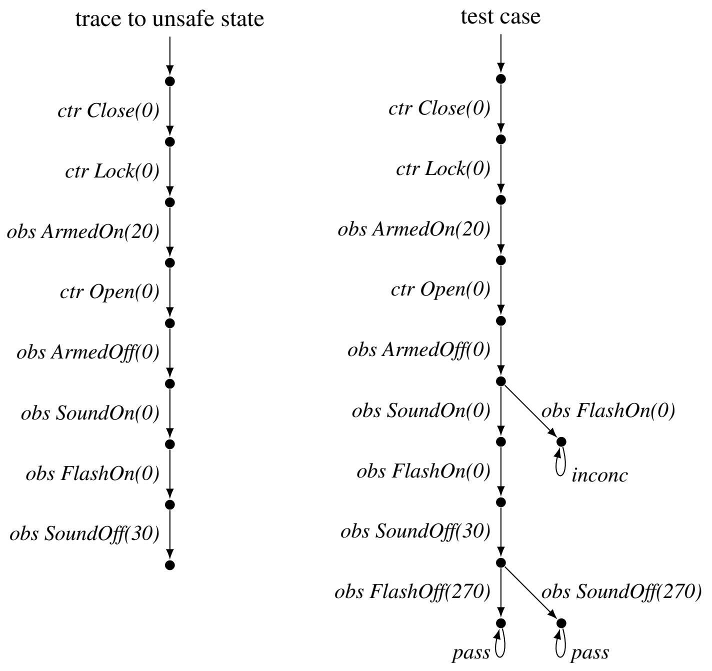
Figure 8.1: LTS of the CAS action system shown in Figure 5.1.

this state needs to be further explored, this means that all its individual action system states are explored and again, the resulting transitions are determinised as described above.

Ulysses, the enumerative test case generation backend of MoMuT::UML, performs the ioco check between two action systems in a similar way. It explores both systems in parallel and performs the determinisation and the check for ioco violations on the fly [50]. However, Ulysses does not use constraint solving to determine possible parameter valuations, but is based on trial and error. It enumerates all possible values for parameters and then tests whether these values fulfil the guard of a given action. For large domains, this is inefficient and our constraint-based approach usually performs better.

Example 8.2. Figure 8.2 illustrates the construction of a test case for the CAS action system shown in Listing 5.1. The left-hand side depicts a trace that leads to an unsafe state. It closes and locks all doors. After 20 seconds, the system gets armed. Then the car is opened, which causes the system to become disarmed. Furthermore, the optical and the acoustic alarms are activated. After 30 seconds, the sound is turned off and the unsafe state is reached.

The test case resulting from the given trace is shown in the middle of Figure 8.2. It is constructed by exploring the action system from the initial state. The LTS representing the full CAS behaviour modelled in the action system is depicted in Figure 8.1. Those parts that are actually explored during test case construction using the given trace are highlighted in blue. In each state that is followed by an



test case in Aldebaran format

```
des(0, 14, 12)
(0, "ctr Close(0)", 1)
(1, "ctr Lock(0)", 2)
(2, "obs ArmedOn(20)", 3)
(3, "ctr Open(0)", 4)
(4, "obs ArmedOff(0)", 5)
(5, "obs FlashOn(0)", 6)
(6, "inconc", 6)
(5, "obs SoundOn(0)", 7)
(7, "obs FlashOn(0)", 8)
(8, "obs SoundOff(30)", 9)
(9, "obs FlashOff(270)", 10)
(10, "pass", 10)
(9, "obs SoundOff(270)", 11)
(11, "pass", 11)
```

Figure 8.2: Construction of a test case for the CAS.

observable action in the trace, additional observable actions found in this state during exploration are added to the test case. In our example, this is the case for the activation of the alarms, which may happen in arbitrary order. Hence, after the *ArmedOff* action, the system may either first turn on the sound or it may first turn on the flash. The action *SoundOn* is already part of the test case. We add a transition labelled by *FlashOn* leading to an inconclusive verdict, which is represented by a self-loop labelled with "inconc". There are no other states in the trace where additional outputs add a new transition leading to an inconclusive verdict. However, we still have to add pass verdicts. In our example, two observable actions are enabled in the unsafe state: *FlashOff* and *SoundOff*, which is turned off a second time in this model. Both lead to pass verdicts represented by self-loops labelled with "pass". Implicitly, all other observable actions not specified in the test case lead to fail verdicts. Note that we do not fully explore the action system: only the blue parts in Figure 8.1 are explored for this trace. In verdict states, the exploration is truncated. Furthermore, alternative input actions do not need to be considered. □

According to Definition 8.1, ioco test cases are LTSs. An LTS can be represented in many different ways, e.g., as a graph as in Figure 8.2. A simple textual format for LTSs is the so-called Aldebaran format[8] originally used in the CADP toolbox[9]. It is furthermore supported by many other tools relying on LTSs like the ioco-based testing tool JTorX[10] or MCRL2[11], which is a formal specification language with an associated toolset for modelling, verification, and validation of concurrent systems. Also MoMuT::UML adopted this format to represent the generated test cases. Hence, also our produced test cases adhere to this format.

Example 8.3. Reconsider Example 8.2, where we constructed the test case depicted in the middle of Figure 8.2. Its textual representation in the Aldebaran format is shown at the right-hand side of Figure 8.2. Each state of the LTS is represented by a natural number. The first line in an Aldebaran file is the so-called *descriptor*. It states general information about the LTS: the initial state (represented by 0), the

[8] `http://www.inrialpes.fr/vasy/cadp/man/aldebaran.html#sect6` (last visit 2014-04-18)
[9] `http://cadp.inria.fr` (last visit 2014-04-18)
[10] `http://fmt.ewi.utwente.nl/tools/jtorx` (last visit 2014-04-18)
[11] `www.mcrl2.org` (last visit 2014-04-18)

number of transitions (14 in our example), and the number of states (12 in our example). Each of the following lines represents one transition and consists of the start state, the label, and the target state. For example, the first transition starts in the initial state 0, is labelled by the input action (prefix *ctr*) Close with parameter 0, and leads to the state represented by the number 1. □

Test cases as the one shown in Figure 8.2 are sometimes denoted as *linear* test cases.

Definition 8.2 (Linear Test Case)
A linear test case contains exactly one path to the unsafe state. Since a model's behaviour may branch, an observation may lead away from the linear path, which is marked by an inconclusive verdict. When executed, a linear test case may result in an inconclusive verdict although it is still possible to reach the unsafe state by an alternative path.

Alternatively, *adaptive* test cases can be generated. For example, MoMuT::UML's existing test case generation backend Ulysses generates adaptive test cases [10].

Definition 8.3 (Adaptive Test Case)
An adaptive test case integrates several paths to the unsafe state into one test case. It only gives an inconclusive verdict if it is impossible to reach the unsafe state. An adaptive test cases may be cyclic.

Despite their advantages, adaptive test cases are harder to handle in test drivers due to potentially cyclic behaviour. This is often not desired by industry, so we decided to generate linear test cases.

As already explained in Chapter 4, the test cases generated by model-based testing are on the same level of abstraction as the test model. Hence, they are often referred to as abstract test cases (Definition 4.3). In order to be executed on the SUT, they have to be concretised, i.e., brought to the level of abstraction of the SUT (cf. Figure 4.2). In Chapter 11, we will run through the whole model-based mutation testing process (cf. Figure 4.1) including concretion and execution of the abstract test cases for the CAS and the particle counter use cases. In the following, we focus on test case generation for our two use cases.

8.2 Experimental Results

Our test case generator, which combines our most optimised refinement checker presented in the previous chapter and the test case construction described above, was applied on the CAS and the particle counter models already used in our previous experiments.

8.2.1 Car Alarm System

We continue our experiments presented in Section 7.2, which indicated that the *ffc-up* constraint solver setting is a reasonable choice for the CAS. Hence, we also use this setting for test case generation. We used Algorithm 7.3 to determine traces to unsafe states, from which we construct test cases. For each of the CAS versions, our refinement check reported 30 conforming mutants ($\sim$15%) up to the maximum exploration depth of the system, which is 13. For each of the remaining non-refining mutants, one test case is generated, i.e., we have 177 test cases. Thereof, 158 are duplicates of others (89%) and 19 unique test cases remain. Figure 8.3 gives an overview of the lengths of these 19 test cases. We measure the length of a test case as the maximum number of consecutive actions in the test case. Hence, the length of a test case does not necessarily correlate with the number of transitions in the test case. For non-deterministic models like the CAS, additional transitions are required for actions leading to inconclusive verdicts. For example, the test case shown in Figure 8.2 has a length of 9, but it consists of 11 transitions

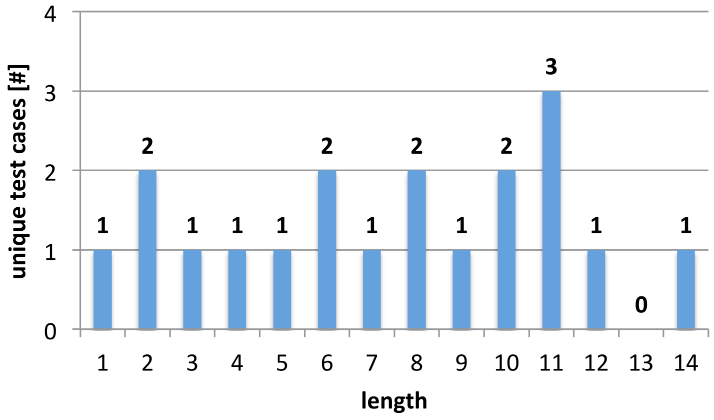


Figure 8.3: Overview of the lengths of the unique test cases for the CAS.

	CAS_1		CAS_10		CAS_100		CAS_1000	
	tc constr.	total	tc constr.	total	tc constr.	total	tc constr.	total
computation time [sec]	1.67	3.65	1.4	3.48	1.6	3.59	1.52	3.45

Table 8.1: Computation times of our test case generator for the CAS case study. The constraint solver setting was *ffc-up* (most-constrained heuristic/ascending value selection).

(excluding pass/inconclusive loops). Figure 8.3 looks the same for all four CAS versions. The longest test case consists of 14 consecutive actions. Hence, an unsafe state has been identified at the maximum depth of 13. As explained above, our test case is one step longer than the trace to the unsafe state in order to check that the implementation only shows specified outputs in the unsafe state.

Table 8.1 gives the computation times required for test case generation for all four CAS versions. Column "*tc constr.*" states the time needed for the test case construction for all 207 mutants. It is approximately 1.5 seconds for each CAS version. Column "*total*" gives the overall time needed for test case generation (refinement check plus test case construction).

As can be seen from these experiments, our test case generation is fast for the CAS case study, but it results in a large amount of test cases for a system of that size. Anyway, most of them are duplicates and can simply be removed by file comparison tools. We present a technique to avoid this high redundancy in the generated test suites in Section 11.1.1.

8.2.2 Particle Counter

Table 8.2 contains the runtimes achieved for test case generation for the particle counter. We used the same action system and model mutants as for evaluating our refinement checker in Section 7.3. In these earlier experiments, we achieved good results with the solver configurations *ffc-up* and *leftmost-down*. Hence, we also used these two configurations for test case generation. The time needed for test case construction of all test cases is approximately 26 seconds for the *ffc-up* configuration and 34 seconds for the *leftmost-down* configuration. The overall test case generation time including the time required by our most optimised refinement checker amounts to 2.2 minutes for *ffc-up* and 2.6 minutes for *leftmost-down*. Note that we do not state maximum values for the total time as it includes the time required for finding the states, which is only performed once for all mutants. The test case construction consumes little runtime compared to the refinement check.

Similar to the CAS case study, the number of generated test cases is rather high and the resulting test suite contains a lot of duplicates. For the particle counter, 121 of the mutated action systems refined the

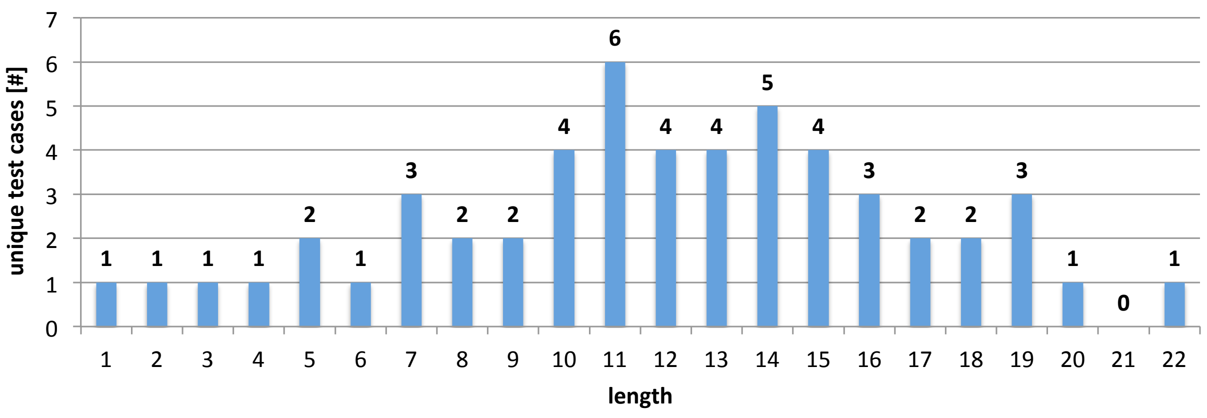


Figure 8.4: Overview of the lengths of the unique test cases for the particle counter.

	ffc-up					*leftmost-down*				
	tc constr.			total		tc constr.			total	
	Σ	ϕ	max	Σ	ϕ	Σ	ϕ	max	Σ	ϕ
computation time	**25.8**	0.04	0.1	**2.2 min**	0.2	**34.3**	0.05	0.1	**2.6 min**	0.2

Table 8.2: Computation times of our test case generator for the particle counter. All values are given in seconds unless otherwise noted.

original (18%). For each of the remaining 551 non-refining mutants, one test case has been generated. Thereof, 53 test cases remain after removing 498 duplicates (90%). The lengths of the 53 unique test cases are shown in Figure 8.4. The longest test case consists of 22 consecutive actions. Note that both constraint-solver settings result in the same test cases.

8.2.3 Comparison of Results

To have at least a weak reference point for our performance, we have also utilised MoMuT::UML's explicit test case generation backend Ulysses [10] to generate tests for our two use cases. This comparison is not totally fair for two reasons. First of all, Ulysses uses a different conformance relation. While we use our rather strict relational refinement for action systems, Ulysses uses ioco, which was already presented in Section 3.2. Furthermore, the generated test cases differ. While our refinement-based implementation generates linear test cases (Definition 8.2), Ulysses creates more complex, adaptive test cases (Definition 8.3).

Table 8.3 summarises the runtimes needed for test case generation with both tools for the CAS and the particle counter models. Both tools were run with the same exploration depths (13 for the CAS models and 28 for the particle counter). Our most optimised test case generator based on refinement needs less than 4 seconds to process all 207 mutants for each CAS version. Ulysses needs 1.7 minutes for CAS_1. For CAS_10 (ten times larger parameter domains), Ulysses runs into massive problems. The

	CAS_1	CAS_10	CAS_100	CAS_1000	particle counter
Ref.-based TCG	3.65 sec	3.48 sec	3.59 sec	3.45 sec	2.2 min
Ulysses	1.7 min	8.8 h	-	-	40.1 h

Table 8.3: Comparison of the test case generation times of our most optimised refinement checker (row *Ref.-based TCG*) and the explicit ioco checker Ulysses for the four CAS versions and for the particle counter.

execution time drastically increases to almost 9 hours. Like our refinement checker, also Ulysses shows outliers. For CAS_10, Ulysses can process 75% of the mutants in ≤ 6 seconds per mutant, while the average mean per mutant is 2.7 minutes. One mutated model is a particular outlier as it caused a runtime of 2.6 hours. We observed a memory usage of up to 6 GB of RAM. We suspect that a significant amount of the execution time is spent on swapping. For CAS_100 and CAS_1000, we did not run Ulysses as the runtimes would be even higher.

We also applied Ulysses to the particle counter model. While our implementation needs only 2.2 minutes for all 672 mutants, Ulysses requires more than 40 hours for the same set of mutants. Due to this long computation time, we transferred this experiment with Ulysses to a server machine different from the PC where all the other experiments were conducted. The server has two 2.5 GHz quad-core processors and 32 GB RAM. Still, in one case, the test case generation for one particular mutant was aborted due to a lack of memory. Processing this mutant took 1.3 hours until it ran out of memory and is included in the total runtime of 40 hours. This was also the maximum amount of time for processing one single mutant.

Our experiments showed that our refinement checker is efficient in terms of runtime. However, it produces a large set of test cases that contains many duplicates. The reduction of this high redundancy in the generated test suites will be addressed in Section 11.1.1.

The comparison with Ulysses showed that our refinement checker is faster by several orders of magnitude. However, the quality of the generated tests is not the same. Ulysses relies on ioco, which is solely based on visible actions. In contrast, our refinement relation also incorporates the states. Hence, if a mutated model reaches a deviating state, this immediately causes non-refinement and a test case is generated. However, if this state discrepancy does not involve a different output action, then the generated test case will not be able to distinguish the mutated from the original model and the fault coverage promised by model-based mutation testing (cf. Chapter 4) may not be guaranteed any more. The reason is that test cases only incorporate action labels and no state information. For model-based testing, which is a black-box testing technique (cf. Definition 2.20), this makes sense as internals like the state of the implementation are not accessible and hence not observable. We enhance our test case generator in Chapter 10 to avoid the generation of non-distinguishing test cases. Prior to that, we generalise our refinement-based test case generator to be able to cope with complex action systems, which is necessary for the integration of our backend into the MoMuT::UML tool chain.

9 Integration into the MoMuT::UML Tool Chain

In the previous chapters, we presented our refinement-based test case generation approach for action systems, which we implemented in Prolog. As already mentioned in the introduction, our test case generation tool shall be integrated into the MoMuT::UML tool chain developed by AIT Austrian Institute of Technology Vienna and colleagues at Graz University of Technology (cf. Section 1.5.1). Therefore, it was necessary to enhance our test case generator to support action systems produced by MoMuT::UML. Before we discuss the required extensions, we give an overview of MoMuT::UML.

9.1 MoMuT::UML

The first version of MoMuT::UML was developed in the MOGENTES project and revised for further usage in the TRUFAL project. MoMuT::UML implements model-based mutation testing for UML models. Figure 1.2 in the introduction gives an overview of the inputs, outputs, and the architecture. As input, it requires (1) a UML class diagram and (2) a UML state machine modelling the SUT. For example, a possible input model for the CAS has already been used in the introduction to describe the functionality of the CAS. The class diagram is depicted in Figure 1.3 and the according state machine is shown in Figure 1.4. There exist plenty of UML modelling tools. Most of them work with a very specific or even proprietary format. Hence, it is not feasible to support all UML modelling tools and MoMuT::UML focuses on Papyrus MDT (an Eclipse Kepler plugin)[12] and Visual Paradigm for UML 10.2[13]. Like all model-based testing tools, MoMuT::UML delivers abstract test cases (Definition 4.3). As already described in Chapter 8, abstract test cases generated by MoMuT::UML are Labelled Transition Systems (LTSs) and are specified in the Aldebaran format. As can be seen from Figure 1.2, MoMuT::UML's architecture distinguishes between *frontend* and *backend*.

9.1.1 Frontend

The frontend deals with model transformations that bring the input model into a format suitable for the backend, i.e., the actual test case generator. To give the UML model a precise formal semantics, it is transformed into a labelled and Object-Oriented Action System (OOAS) [140], which we briefly described in Section 5.4.1. Most UML elements can be directly mapped to the corresponding OOAS structures, e.g., classes, member fields, and methods. Transitions of the state machine are roughly speaking mapped to actions. Only the time- and event semantics of UML needs to be expressed by more complex OOAS structures. This transformation is implemented in the *UML2OOAS* component (cf. Figure 1.2). Subsequently, the OOAS is transformed into a complex action system as introduced in Section 5.2.3. Note that during this transformation, the parallel composition of the individual components is resolved and as a result, one single action system comprises the whole modelled functionality. The transformation from OOASs into action systems is implemented in the *OOAS2AS* component (cf. Figure 1.2) and has been sketched in Section 5.4.1.

Furthermore, the frontend is responsible for the injection of faults into the original model to construct a set of model mutants as required for model-based mutation testing (cf. Chapter 4). The mutation operators are defined on the UML level and are implemented in the UML2OOAS component. The mutation operators are applied to the following state machine elements: triggers, guards, transition effects, as well as entry- and exit actions. The elements are either removed or replaced with another element of the same

[12] `https://www.eclipse.org/papyrus` (last visit 2014-04-18)
[13] `http://www.visual-paradigm.com/product/vpuml` (last visit 2014-04-18)

type from the model. This leads to $O(n)$ mutants for the removals and $O(n^2)$ mutants for the replacements (with n being the number of corresponding elements in the model). Additional mutation operators exist for change trigger expressions, for guards expressed in Object Constraint Language (OCL)[14], as well as for effects, entry/exit actions, and method bodies (expressed in the Activity and Guard Specification Language (AGSL) [85]). The modifications made here are the exchange of operators or the modification of literals. Furthermore, (sub-)expressions are replaced by true/false or negated. They all lead to $O(n)$ mutants. After all model mutants have been generated, they are converted into action systems similarly as the original UML model.

The UML2OOAS component was developed by AIT Austrian Institute of Technology Vienna. The OOAS2AS component was developed by Willibald Krenn at Graz University of Technology and revised in the course of his occupation at AIT Austrian Institute of Technology Vienna. For details on the transformations in the frontend, we refer to Krenn et al. [140]. The following project deliverables date back to the MOGENTES project and describe the mutation operators (Deliverable D3.1b [86]) and the supported UML modelling language features including OCL and AGSL (Deliverable D3.2b [85]). The basics described in these deliverables still apply. However, some details changed since then due to technical reasons. For example, the UML editor Papyrus UML, which had been used in MOGENTES, has undergone a substantial redesign including a modified file format specification. Hence, also MoMuT::UML's frontend had to be adapted to support the up-to-date modelling tool, which is called Papyrus MDT and is an Eclipse plugin.

9.1.2 Backend

The action systems created by the frontend serve as input for the backend – the actual test case generation engine. In the MOGENTES project, the tool *Ulysses* has been developed to generate mutation-based test cases. It explores a given original and a mutated action system in parallel yielding their LTS semantics. For example, the LTS representing the behaviour of the CAS action system shown in Listing 5.1 has already been depicted in Figure 8.1. Ulysses uses the Input-Output Conformance (ioco) relation already introduced in Section 3.2. For the ioco check, it performs the determinisation of the LTSs and the check for ioco violations on the fly, i.e., it does not fully explore both action systems, but stops if an unsafe state is found. If the mutated action system is not ioco-conform to the original action system, a test case is generated that distinguishes the original from the mutated model. As already described in Section 8.1, these test cases are adaptive (Definition 8.3), i.e., they incorporate all possible paths to a given unsafe state. For further information on the underlying theory and techniques, we refer to the following publications [10, 7, 9, 8, 50].

Ulysses does not only support complex action systems as described in Section 5.2.3. It additionally allows for complex data types like lists. With the Ulysses backend, MoMuT::UML uses lists. However, for our constraint-based approach, MoMuT::UML offers an alternative transformation to action systems without lists. Another feature of Ulysses is its support for *qualitative* action systems [11, 50], which allows for generating test cases for hybrid systems (cf. Section 5.4.2). However, this functionality is not incorporated by MoMuT::UML.

Like our refinement-based test case generator, Ulysses is also implemented in SICStus Prolog. It performs the exploration of the action system's state spaces in a similar way as we explore the original action system for test case construction (cf. Chapter 8). However, Ulysses only uses a constraint-based approach for qualitative reasoning over continuous behaviour, which is not incorporated in complex action systems as generated by MoMuT::UML's frontend. For the discrete behaviour of our complex action systems, Ulysses does not use constraint solving to determine possible parameter valuations, but is based on trial and error. It enumerates all possible values for parameters and then tests whether

[14]`http://www-st.inf.tu-dresden.de/ocl` (last visit 2014-04-18)

these values fulfil the guard of a given action. For large parameter domains, this is inefficient and our constraint-based approach usually performs better (cf. Section 8.2.3). However, our constraint-based refinement checker described in the previous chapters is not able to process complex action systems and hence cannot interact with MoMuT::UML's frontend. In the next section, we discuss how we adapt our implementation for compatibility with MoMuT::UML's frontend.

9.2 Required Extensions

To integrate our refinement-based test case generator into MoMuT::UML, we had to add support for complex action systems, which extend plain action systems as described in Section 5.2.3. In the following, we explain the effects of each language extension on our implementation.

9.2.1 Class Data Types

Complex action systems introduce an enumeration data type, which is used to represent classes as lists of object identifiers. For the usage with constraints, our implementation maps each object identifier to an integer value – similarly as we already map action labels to integers. We use consecutive integer values such that we can state the domain of the data type as one interval. For example, consider a data type $myClass$ defined as $\texttt{type}(myClass, X) :\!-\, member(X, [obj1, obj2, ob3])$. We map each object identifier to an integer, e.g., $obj1$ to 1, $obj2$ to 2, and $obj3$ to 3. The type can then be defined as an integer between 1 and 3.

9.2.2 Methods

As already explained in Section 5.2.3, method calls are inlined. In order to translate a method call into constraints, we need the method's definition, i.e., its head and body. Parameters are passed by Prolog's unification, i.e., the actual parameters from the method call are unified with the formal parameters in the method's head. This is possible since formal parameters are represented by Prolog variables. Actual parameters are also either Prolog variables or constants. By this unification, the formal parameters are also *"substituted"* by the actual parameters in the method's body, which can now directly be translated into constraints.

To determine the required method definition, we have to consider that the translation of OOASs into our complex action systems creates one method definition per object. Each of these definitions directly refers to the object's member variables, which are state variables of the action system. Both, the state variables and the methods belonging to a specific object are prefixed by the object's identifier. Remember that this is possible since there is no dynamic creation or destruction of objects in MoMuT::UML's OOASs (cf. Section 5.4.1). In a method call, the operator $\backslash$ is used. For example, $obj1\backslash foo(3)$ calls the method foo for object $obj1$ with parameter 3. The according method definition is identified by $obj1\_foo$, i.e., the backslash operator is replaced by an underline. In our predicative semantics, we assumed a lookup function $getB$, which takes this method identifier and returns the method's body. Implementing this lookup function is straightforward. Note that parameters are not considered for finding the definition of a method, i.e., we do not support overloading of methods.

Additionally to methods that are called with a given object identifier, methods may be called on a state variable containing an object. For example, $v\backslash foo(2)$ calls the method foo with parameter 2 on the object assigned to the action system variable v. In such a case, we do not know which method definition we have to use as we cannot determine at compile time which object will be assigned to v at runtime. Hence, we lookup the data type of the variable v and consider each possible object obj of this class. In Figure 5.2, this lookup function was denoted as $getClass$. The constraint representing a method call on a state variable is a disjunction over all instances of this method and each method is guarded by a test whether the variable v currently holds the corresponding object (cf. Figure 5.2). Note that we do not need to perform a type check as this is already handled in the OOAS2AS component of the frontend.

Finally, our syntax allows for plain procedure calls in the form $\backslash m$. Its implementation is straightforward as the procedure name m directly serves as identifier for the definition. Plain procedure calls do not occur in complex action systems that are created by MoMuT::UML. However, they are convenient for manual modelling.

9.2.3 Prioritising Composition

Complex action systems introduce an additional operator for composition, which gives priority to the left-hand side operand. It can be applied both in action bodies as well as for composing actions in the do-od block. $B_1 \ // \ B_2$ states that if B_1 is enabled, B_2 must not be enabled. In our semantics, this is expressed as $\phi(B_1) \vee (\neg\phi(B_1) \wedge \phi(B_2))$ and can be directly used in our constraint representation.

However, we have to take care for the prioritising composition of actions with parameters. Remember that our translation of actions into constraints uses one set of variables $\overline{P}$ to encode action parameters in both the mutated and the original action system (cf. Section 6.3.1, especially Example 6.1). In this way, the parameters between the original and the mutated action system are synchronised for the refinement check. However, for determining the enabledness of B_2, we need to hide, i.e., existentially quantify the parameters of B_1 in its negation. Hence, we use the original semantics for actions with parameters (cf. Figure 5.2) where parameters are local, i.e., existentially quantified. Note that this quantification is essential due to the required negation. A similar issue has already been discussed with respect to sequential composition (cf. Section 6.4.2, Pitfall 2).

9.2.4 Sequential Composition in the do-od Block

For one iteration of the do-od block of a plain action system, it was sufficient to use one variable to represent the chosen action and one vector of variables representing the parameters of this action (cf. Section 6.3.1). Due to the introduction of the sequential composition operator in the do-od block, we have to consider that sequences of actions may be executed in one iteration. Therefore, we analyse the do-od block to determine the greatest possible number n of consecutive action calls and use n variables in our constraints to represent these actions. Furthermore, we use n variable vectors representing the parameters of these actions. For parameters in plain action systems, we already introduced a special integer representing that a parameter variable is unused (cf. Section 6.3.1). We apply the same concept for unused action variables and parameter variables in the traces as we again have to deal with varying numbers of parameters and consecutive actions. For example, the do-od block $(a \ ; \ b) \ [] \ c$ may either execute the two actions a followed by b or it may execute only one action, i.e., action c. In the latter case, the second action variable is unused.

Note that sequential composition hides intermediate states (cf. Figure 5.2). Hence, states between actions that are sequentially composed in the do-od block are not observable from the outside. The only points at which states are observable are before and after the execution of an iteration of the do-od block. This causes a difference to the LTS semantics used by Ulysses, which affects non-deterministic choices of sequential compositions in the do-od block. We illustrate this difference in the semantics by the following example.

Example 9.1. Consider an action system where the state consists of one variable s, which is an integer between 0 and 3. Let the initial state be $s = 0$. Furthermore, the following three actions are defined:

$$a :: (s = 0) \Rightarrow (s := 1) \qquad b :: (s = 1) \Rightarrow (s := 2) \qquad x :: (s = 1) \Rightarrow (s := 3)$$

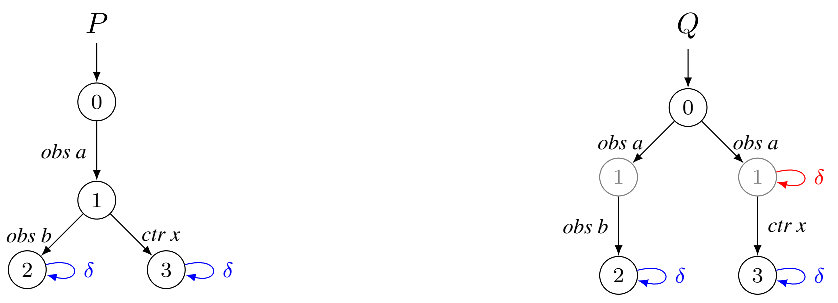


Figure 9.1: Semantic difference between Ulysses and our ioco check.

Let a and b be output actions and x be an input action. Furthermore, let the do-od block be defined as a ; $(x \,[\,]\, b)$, i.e., the action a is sequentially composed with the non-deterministic choice of x and b.

Figure 9.1 shows the LTS representation P of this action system as generated by Ulysses on the left-hand side. The labels of the LTS states correspond to the valuation of the state variable s. The LTS Q that is derived from the same action system by our constraint-based approach is depicted on the right-hand side. The LTS Q splits the intermediate state valuation $s = 1$ into two separate LTS states. This is due to the fact that intermediate states of sequential compositions are unobservable. Hence, our constraint-based approach does not know about the intermediate state valuation $s = 1$, which is indicated by the gray colour of these states.

In our implementation, one iteration of the do-od block is encoded in constraints, which are repeatedly solved using varying pre-states. For the initial state $s = 0$, the constraint solver returns two successor states: $s = 2$ with trace $\langle a, b\rangle$ and $s = 3$ with the trace $\langle a, x\rangle$. From this information, it cannot be determined whether the action a leads to the same state or to different states in the two traces. In our implementation, we assume different states. As a consequence, additional quiescence (Definition 3.16) is introduced: the LTS state with label 1 on the right-hand side is quiescent as it only has one outgoing transition, which is an input. In contrast, the LTS P is not quiescent in state $s = 1$ as $1 \xrightarrow{b}$ and $b \in L_O$. Note that if we merge the intermediate states, we possibly face the opposite problem, i.e., we could miss quiescence. □

This discrepancy only arises if the intermediate state is a mixed state, i.e., it has outgoing input as well as outgoing output/internal transitions. This semantic difference has to be considered during modelling. In the given case studies, it has not been an issue.

9.2.5 Internal Actions

In complex action systems, each action may be hidden by enclosing it in $i(...)$ in the do-od block. In the LTS semantics used by Ulysses, internal actions are mapped to τ, which denotes an internal action in LTSs (cf. Definition 3.12). For our constraint representation, we map action labels to integers. For internal actions, we introduce an additional integer for encoding the τ-event. A hidden action may have parameters. However, they are not observable, i.e., the resulting τ-action does not have parameters. Like for the prioritising composition of actions in the do-od block, we use the original semantics for actions with parameters (cf. Figure 5.2) where parameters are local, i.e., existentially quantified, to hide them.

Internal actions have the following consequences for our refinement relation. If the mutant performs an internal action, it must also be allowed by the original. Note that in case of consecutive internal actions, it must be exactly the same number of internal actions.

9.2.6 Integration of the SMT Solver Z3

For the prioritising composition of actions and internal actions, we need existential quantifications to hide parameters. However, explicit quantifications are not supported by the used constraint solver. We faced a similar problem with sequential composition (cf. Section 6.4.2, Pitfall 3). Our solution was to rewrite our actions into a normal form that allows for the application of the one-point rule to eliminate the existential quantifications (cf. Section 6.4.2). We can neither apply this solution to the prioritising composition of actions, nor to internal actions. In these cases, we need to quantify over parameters, which are not assigned to one single value, but are constrained arbitrarily in the guards of actions. Hence, it is in general not straightforward to eliminate the required quantifiers and we decided to rely on an SMT solver that supports a theory for quantifiers. Furthermore, most modern SMT solvers directly support incremental solving, which we had to emulate with the used constraint solver by backtracking in Prolog (cf. Section 7.1.4). We decided to use Microsoft's SMT solver Z3 (version 4.3.1), which supports both quantifiers and incremental solving.

We prepare the constraints as SMT-LIB v2 [35] formulas. This facilitates the possible integration of other SMT solvers. We interface with Z3 via its C-API, which we encapsulated in SICStus Prolog as a *foreign resource*. Therefore, each C function needs to be mapped to a Prolog predicate. In particular, the data types of the parameters and return values need to be specified. Further required *glue code* is automatically generated by SICStus Prolog [88].

By default, Z3 does not apply all of its implemented quantifier elimination tactics. We experienced that the simple eliminations that are performed by default are not sufficient for our type of constraints. Some quantifiers have been instantiated, which caused the solving time to become unreasonable. Hence, we use Z3's *"qe"* tactic, which performs all of Z3's quantifier elimination techniques that preserve equivalence for the given formula. This is not straightforward in combination with incremental solving. In Z3, a solver that applies the quantifier elimination tactic can be created. However, it will not be incremental. Our solution is to use the quantifier elimination tactic as a pre-processing step, i.e., we use the tactic to rewrite our formulas and pass the resulting quantifier-free formulas to an incremental solver instance of Z3.

Note that all of the following experiments in this thesis are conducted using the SMT solver Z3.

9.3 Complexity of Action Systems Generated by MoMuT::UML

We experienced that action systems automatically generated by MoMuT::UML are more complex than manually designed action systems. This is not a surprise, however the actual dimension is astonishing. We compared manually designed plain action systems (cf. Section 5.2.2) with complex action systems (cf. Section 5.2.3), which were generated by MoMuT::UML from UML models, with the prerequisite that both models specify exactly the same visible behaviour. Therefore, we used the CAS as well as the particle counter use case. In the following, we denote the plain action system directly modelling the CAS use case as *CAS_AS*. Parts of this action system have already been shown in Listing 5.1. Although it models the same system, it is not exactly the same as the action system of the CAS that was used in our previous experiments. As already stated in Section 6.6.1, the action systems differ in the way time is modelled. The complex action system generated from the UML model by MoMuT::UML is referred to as *CAS_UML*. Analogously, we name the plain action system modelling the particle counter use case *PC_AS* and the complex action system model generated by MoMuT::UML is called *PC_UML*.

Table 9.1 states model metrics on the action system level for both use cases and both modelling approaches. For the CAS use case, CAS_AS comprises 10 actions, while CAS_UML defines 51 actions (41 of them are internal actions). The approximate complexity of the models can be estimated by considering the types of the state variables, which are either Boolean or bounded integers. Theoretically, the 6 state

	CAS_AS	CAS_UML	PC_AS	PC_UML
actions [#]	10	51	26	109
state variables [#]	6	35	10	74
possible states [#]	1600	$1.7 \cdot 10^{18}$	$1.6 \cdot 10^{9}$	$1.2 \cdot 10^{31}$
reachable states [#]	19	229	1725	$> 850\,700$
required exploration depth	11	17	28	> 25
time needed for exploration	~ 1 sec	~ 3 sec	~ 12 sec	> 4 days

Table 9.1: Metrics for the CAS and particle counter (PC) use cases modelled either directly as an action system (AS) or modelled in UML and automatically transformed into an action system by MoMuT::UML.

variables of CAS_AS span a state space of 1600 states, while CAS_UML comprises 35 state variables that theoretically build a state space of $1.7 \cdot 10^{18}$ states. However, not all of these states are reachable from the initial system state. In CAS_AS, 19 states are actually reachable. The depth in terms of the number of consecutive visible actions required to reach all of these states is 11. In CAS_UML, 229 states are reachable. The required exploration depth is 17. The computation time for exploring the state space up to these depths is 1 and 3 seconds respectively.

For the particle counter use case, PC_AS consists of 26 actions, while PC_UML defines 109 actions (83 thereof are internal). PC_AS uses 10 state variables that theoretically build a state space of $1.6 \cdot 10^9$ states from which 1725 are actually reachable. The depth to find all reachable states is 28. The exploration needed approximately 12 seconds. Again, the state space of the automatically generated action system is several orders of magnitude larger than the state space of the manually designed plain action system model. PC_UML requires 74 state variables resulting in a state space of $1.2 \cdot 10^{31}$ states from which more than 850 000 states are reachable. We can only give a lower bound for the number of actually reachable states as it was not possible to fully explore this model. The exploration up to depth 25 took approximately 4 days. A deeper exploration would require even longer runtimes. Therefore, we stopped at depth 25.

These metrics illustrate that the automatically generated action systems are several orders of magnitude more complex compared to manually written plain action systems that encode exactly the same behaviour. Furthermore, MoMuT::UML's action systems have a special structure that hinders our approach to take full advantage of its optimisations. In particular, we observed that the do-od block of an action system generated by MoMuT::UML usually has a prioritising composition as the outermost operator. Hence, the do-od block is one monolithic construct, which cannot be split into individual actions or at least several compositions of actions. The decomposition of the do-od block was originally used in our refinement check to construct a smaller constraint representing non-refinement (cf. Section 6.3.1). For action systems generated by MoMuT::UML, our non-refinement constraint comprises the whole negated original action system as well as the whole mutated action system. We could circumvent this by rewriting the do-od block to a similar normal form as used for the application of the one-point rule (cf. Section 6.4.2). However, this would not fully resolve the problem as prioritising composition is not distributive. Although

$$A \,//\, (B \,[]\, C) \quad = \quad (A \,//\, B) \,[]\, (A \,//\, C)$$

holds, the other direction does not hold in general [181]:

$$(A \,[]\, B) \,//\, C \quad \neq \quad (A \,//\, C) \,[]\, (B \,//\, C)$$

In the following, we report on test case generation with our refinement checker for both use cases and compare the runtimes between the manually written plain action systems and the automatically created complex action systems. All experiments were conducted on the same PC, which runs a 64-bit Linux

		non-refining	refining	total
mutants [#]		176	79	255
time – find states	Σ	-	-	0.2
time – ref. check	Σ	1.15	0.43	1.58
	ϕ	0.01	0.01	0.01
	max	0.02	0.02	0.02
time – tc constr.	Σ	68.23	-	68.23
	ϕ	0.39	-	0.27
	max	1.33	-	1.33
total computation time	Σ	7.5 min	2.7 min	10.2 min
	ϕ	2.55	2.05	2.39
	max	4.19	3.51	4.19
total computation time without log	Σ	69.44	0.46	69.9
	ϕ	0.39	0.01	0.27
	max	1.34	0.02	1.34

Table 9.2: Test case generation via refinement checking up to depth 20 for the CAS_AS model. All values are given in seconds unless otherwise noted.

(Ubuntu 12.04). It is equipped with 8 GB RAM and an Intel i7 quad-core processor (3.4 GHz). However, our implementation utilises only one core.

9.4 Experimental Results

9.4.1 Car Alarm System

Plain Action System

We mutated the CAS_AS model using the following mutation operators. Setting guards to true resulted in 37 mutations. By swapping the equals and unequals operators, 71 mutations were created. Furthermore, we model off-by-one errors by incrementing each integer constant by one, which resulted in 120 mutants. Finally, inverting Boolean constants caused 27 mutations. Overall, we have 255 mutated models.

Table 9.2 gives an overview of test case generation with our refinement checker for the CAS_AS model. The used exploration depth was set to 20, which covers the full state space of the model as all reachable states were discovered up to depth 11 (cf. Table 9.1). We used all 255 mutated models from which 69% did not refine the original model. In total numbers, 176 model mutants did not refine the original model, whereas 79 model mutants refined the original. The following rows in the table state the computation times for finding all reachable states up to the given search depth, the time required for the refinement check, the time needed to construct test cases from the traces to the unsafe states (cf. Chapter 8), and the total computation time. Note that the total computation time does not include the time for finding the reachable states as this task is only performed once for all mutants (cf. Section 7.1.3) and cannot be split between non-refining and refining mutants. Furthermore, the total computation time is not equal to the sum of the time for the refinement check and the time for test case construction due to input/output operations such as parsing, writing logs, etc. We noticed that writing logs for our results, which mostly concerns recording the runtimes for the individual tasks in a file on the hard drive, takes a considerable amount of time (more than 88% of the total computation time). As this is only required for our analysis and is not needed in practice to generate test cases, we also state the total time without writing this log file. For each category, we state the total time needed to process all 255 model mutants

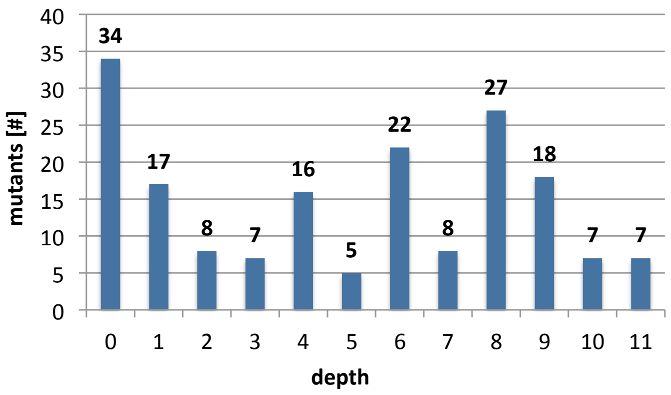


Figure 9.2: Diagram stating how many mutants showed non-refinement at a specific depth for the CAS_AS model.

(Σ), the average value for one mutant (ϕ), and the maximum value for one mutant (max). As can be seen from the table, for this simple example the refinement check is very fast (less than 2 seconds for all mutants) compared to the test case construction (almost 70 seconds in total), which includes writing the generated test case into a file. Furthermore, the refinement check for a refining mutant takes almost equally long as for a non-refining mutant (0.01 seconds on average). Overall, we could not identify any mutant to be a noticeable outlier. Hence, we did not explicitly state values for the median or for quartiles.

Figure 9.2 gives an overview of the depths at which an unsafe state (Definition 6.1) was found. The unsafe states are distributed over all depths. In 34 models, the initial state (depth 0) is an unsafe state, i.e., it either leads to an incorrect next state or it allows an action (possibly including parameters) that is not specified by the original model.

Complex Action System from UML Model

As described above, MoMuT::UML provides a rich set of mutation operators for UML state machines. For our CAS model, the following mutation operators were applied: setting guards to false resulted in 22 mutations, removing AGSL statements caused 13 mutations, and removing entry and exit actions resulted in 1 mutant each. By removing effects from transitions, 7 mutations were created. Further 42 mutations were caused by replacing effects with other effects in the model. Furthermore, 14 mutations were generated by removing signal triggers and 3 mutations were caused by removing time triggers. The replacement of signal events caused 42 mutations, and mutating time events resulted in 17 mutations. In total, 162 mutated models have been created.

Table 9.3 reports on the test case generation with our refinement checker for the CAS_UML model. It has the same structure as Table 9.2. We again report computation times for finding the reachable states, for the refinement check, the test case construction, and the total time with and without writing of log files. The computation times for all 162 mutated models is given by Σ, the average value for one mutant is represented by ϕ, and finally we state the maximum value for one mutant (max). Again, there were no noticeable outliers and we refrained from stating quartiles. The used exploration depth was 20, which again covers the full state space of the model (cf. Table 9.1). The ratio between refining and non-refining mutants is 149 to 13, i.e., for this example most mutants were non-refining (92%). The time required to find all 229 reachable states was 3 seconds. This task is only performed once for all mutated models. Hence, it is not split between non-refining and refining mutants. Again, there is no considerable difference in the average time needed to check for refinement between a non-refining and a refining mutant (0.38 and 0.31 seconds respectively). While the refinement check for CAS_AS required less than 2 seconds for 255 mutants, it took approximately 1 minute for 162 mutants of the CAS_UML

		non-refining	refining	total
mutants [#]		149	13	162
time – find states	Σ	-	-	3
time – ref. check	Σ	56.22	4.09	60.31
	ϕ	0.38	0.31	0.37
	max	0.52	0.44	0.52
time – tc constr.	Σ	67.30	-	67.30
	ϕ	0.45	-	0.42
	max	1.27	-	1.27
total computation time	Σ	7.1 min	33.62	7.7 min
	ϕ	2.87	2.59	2.85
	max	4.06	3.00	4.06
total computation time without log	Σ	2.1 min	4.29	2.2 min
	ϕ	0.84	0.33	0.80
	max	1.72	0.45	1.72

Table 9.3: Test case generation via refinement checking up to depth 20 for the CAS_UML model. All values are given in seconds unless otherwise noted.

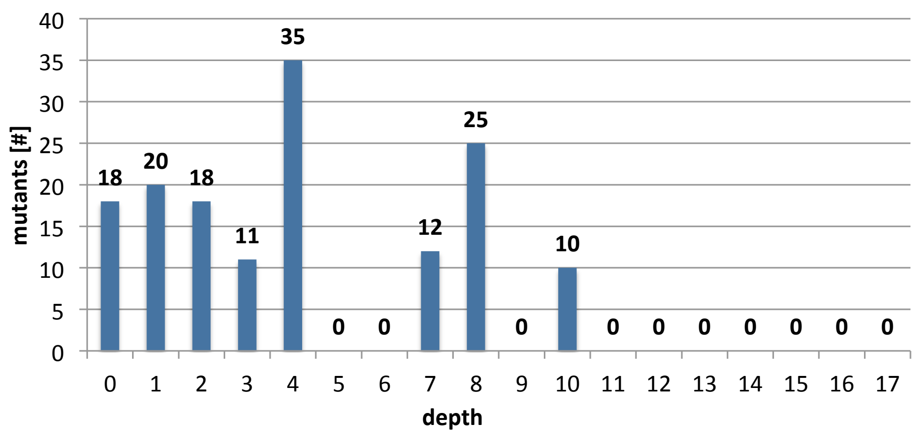


Figure 9.3: Diagram stating how many mutants showed non-refinement at a specific depth for the CAS_UML model.

model. The construction of test cases takes almost the same amount of time (67 seconds). Again, writing our log file requires the majority of the overall computation time (about 5 minutes). Without log file writing, the total computation time for all mutants is 2.2 minutes.

We analysed the depths of the found unsafe states. As can be seen from Figure 9.3, the unsafe states are not located very deep in the model. Although new states could be discovered up to depth 17, the deepest unsafe states are located at depth 10. The lack of unsafe states in depths 5 and 6 is due to the structure of the action system's do-od block and can be explained as follows. We count the depth in terms of consecutive, visible actions. Furthermore, states are only observable before and after one iteration of the do-od block. The analysis of the do-od block of CAS_UML showed that from states at depth 4 it is only possible to move on via a sequence of 3 sequential actions in one do-od block: disarm, turn flash on, turn sound on (the last two actions may happen in any order). Hence, the next state reached is at depth 7 and there are no visible states at depths 5 or 6. The lack of unsafe states at depth 9 has the same explanation. In this case, the actions for turning off the flash and sound are sequentially composed in the do-od block. Why there are no unsafe states located at depths 11 or higher cannot be explained in this way. Instead, a possible explanation is that the used mutation operators did not trigger failures in

		non-refining	refining	total
mutants [#]		573	141	714
time – find states	Σ	-	-	7
time – ref. check	Σ	39.96	59.43	99.39
	ϕ	0.07	0.42	0.14
	max	0.46	0.59	0.59
time – tc constr.	Σ	4.9 min	-	4.9 min
	ϕ	0.52	-	0.42
	max	1.55	-	1.55
total computation time	Σ	25 min	5.9 min	30.9 min
	ϕ	2.62	2.50	2.60
	max	5.32	5.08	5.32
total computation time without log	Σ	5.6 min	1 min	6.6 min
	ϕ	0.59	0.42	0.56
	max	1.86	0.60	1.86

Table 9.4: Test case generation via refinement checking up to depth 30 for the PC_AS model. All values are given in seconds unless otherwise noted.

these depths. However, it is more likely that the mutations cause failures deep in the model, but at the same time they trigger failures in lower depths. As we perform a breadth-first search, we first detect the failures in lower depths and stop our checks.

9.4.2 Particle Counter

Plain Action System

We also applied our test case generator on the particle counter models. We used the same mutation operators to mutate the PC_AS model as we used for the CAS_AS model. Setting guards to true yielded 101 mutations, swapping equality and inequality operators caused 249 mutations, and the incrementation of integer constants by 1 resulted in 178 mutations. Finally, 186 mutations were generated by inverting Boolean constants. In this way, a total of 714 mutated models were created.

Like the two CAS models, we could fully explore the plain action system PC_AS. We set the maximum search depth to 30. New states could be found up to depth 28. Results are listed in Table 9.4. Again, we give the same values as for the CAS models in Table 9.2 and Table 9.3. The computation times for all 714 mutated models is given by Σ, the average value for one mutant is represented by ϕ, and finally we state the maximum value for one mutant (max). Again, there were no noticeable outliers and we refrained from stating quartiles. Like for CAS_AS, the ratio between the runtime of the refinement check (99 seconds) and the test case construction (4.9 minutes) is unbalanced. The time required to log computation times makes up the majority of the overall runtime, whereas the time for finding the reachable states is negligible (7 seconds). Approximately 80% of the 714 mutants do not refine the original model. The average value for the refinement check of one refining mutant is 0.07 seconds, while a non-refining mutant requires 0.42 seconds. We assume that this is due to the fact that this model has more reachable states (1725) than the CAS models (19 and 229 respectively). Hence, the number of checks whether a state is an unsafe state is higher than for the CAS models. Note that for refining mutants all reachable states have to be checked, whereas for non-refining mutants only a fraction of the overall number of states need to be tested. Figure 9.4 gives an overview of the depths at which an unsafe state was found. Like for the CAS_UML model, most unsafe states were discovered in lower depths. Between depth 20 and the full depth of 28, only 1 unsafe state was discovered. As already explained for the CAS_UML

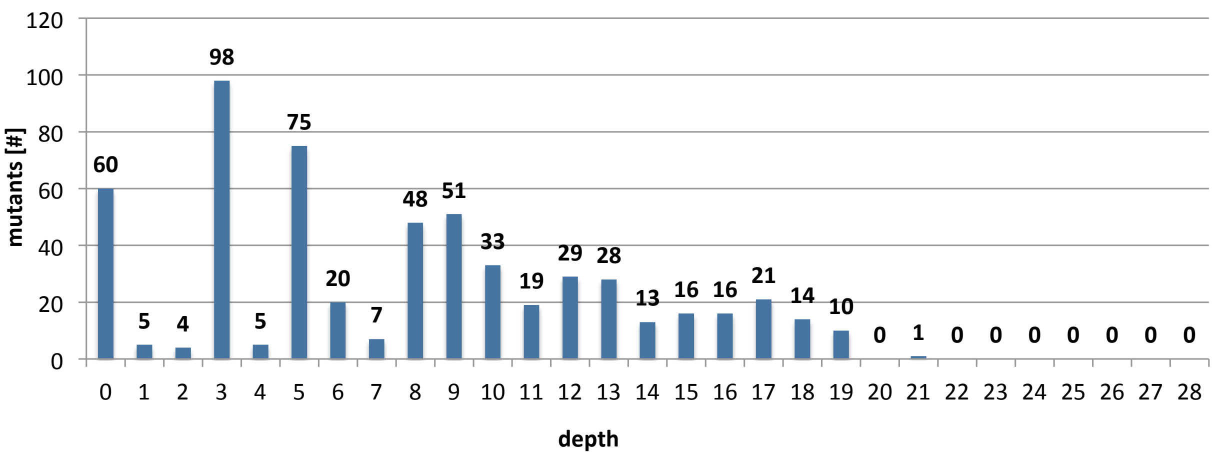


Figure 9.4: Diagram stating how many mutants showed non-refinement at a specific depth for the PC_AS model.

model, we suppose that some mutations in fact cause failures deeper in the model, but at the same time they inject faults in lower depths where we detect them first due to our breadth-first search.

Complex Action System from UML Model

The UML state machine for the particle counter has been mutated with MoMuT::UML yielding 1185 mutated models. The following mutation operators were applied. Guards were set to true and false yielding 39 and 23 mutations respectively. Furthermore, the following elements were removed from the model: AGSL statements (23 mutations), change triggers (1 mutation), effects (14 mutations), entry actions (11 mutations), exit actions (5 mutations), signal triggers (27 mutations), and time triggers (6 mutations). The following elements were replaced by other elements of the same type: effects (182 mutations), entry actions (110 mutations), exit actions (20 mutations), and signal events (506 mutations). Furthermore, time events were mutated (36 mutations), e.g., by incrementation by 1. Finally, the exchange of OCL operators caused 29 mutations, and the mutation of OCL subexpressions yielded 153 mutations.

While both CAS models and PC_AS could be fully explored, we could not cover the complete state space of PC_UML. As already shown in Table 9.1, we explored the model up to depth 25 and there were still states remaining for further exploration. As this took 4 days, we limited our search depth for our test case generation experiments to a depth of 15, which results in more reasonable runtimes as can be seen in Table 9.5. Again, Σ denotes the computation times for all 1185 mutated models, the average value for one mutant is represented by ϕ, and finally we state the maximum value for one mutant (max). For this model, we identified outliers in the refinement check and hence also in the total runtimes. Therefore, we stated also values for the quartiles ($Q_1/Q_2/Q_3$). Finding all reachable states up to depth 15 required almost 13 minutes. However, this is only performed once for all mutated models. Approximately 84% of all 1185 mutants were non-refining up to depth 15. Thereof, 75% could be processed in less than 10 seconds per mutant (cf. Q_3 for the refinement check of non-refining mutants). However, outliers (like the maximum of 3.6 minutes) raised the arithmetic mean value (ϕ) to almost 23 seconds. The median (Q_2) is approximately 6 seconds. For refining mutants the situation is similar. The median is 13.5 seconds, while the arithmetic mean is 100 seconds. The third quartile Q_3 is already 192 seconds per mutant. We checked the data and found out that many refining mutants are identified in around 14 seconds, while many others required about 200 seconds, i.e., there were hardly any values in between.

The analysis of the depths of the unsafe states for the refining mutants led to Figure 9.5. Again, there are depths without any unsafe state. This can be explained by the structure of the action system's do-od

		non-refining	refining	total
mutants [#]		996	189	1185
time – find states	Σ	-	-	12.8 min
time – ref. check	Σ	6.3 h	5.3 h	11.6 h
	ϕ	22.88	100.14	35.20
	$Q_1/Q_2/Q_3$	5 / 6 / 10	12 / 13.5 / 193	5 / 8 / 16
	max	3.6 min	3.8 min	3.8 min
time – tc constr.	Σ	22.4 min	-	22.4 min
	ϕ	1.35	-	1.13
	max	3.96	-	3.96
total computation time	Σ	7.3 h	5.4 h	12.7 h
	ϕ	26.38	102.25	38.48
	$Q_1/Q_2/Q_3$	8 / 9 /13.5	14 / 16 / 195	9 / 11 / 20
	max	3.7 min	3.8 min	3.8 min
total computation time without log	Σ	6.7 h	5.3 h	12 h
	ϕ	24.27	100.18	36.38
	$Q_1/Q_2/Q_3$	6 / 7 / 11.5	12 / 14 / 193	6 / 9 / 18
	max	3.7 min	3.8 min	3.8 min

Table 9.5: Test case generation via refinement checking up to depth 15 for the PC_UML model. All values are given in seconds unless otherwise noted.

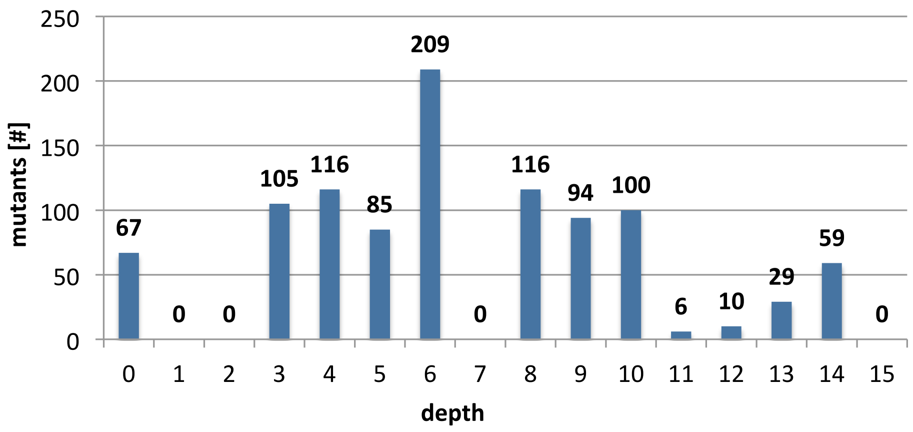


Figure 9.5: Diagram stating how many mutants showed non-refinement at a specific depth for the PC_UML model.

block, which contains sequential compositions of actions that hide states at certain depths – similar as for CAS_UML (cf. Figure 9.3). Note that we could not fully explore the model. Hence, there might be unsafe states with a depth higher than 15, which were missed by our bounded refinement check.

9.5 Discussion

We described how we extended our test case generator in order to support action systems generated by MoMuT::UML's frontend and we compared the complexity of manually created action systems with automatically created complex action systems. Therefore, we calculated a number of metrics for four action system models (Table 9.1). Two action systems represent the same behaviour of the CAS. While one was directly modelled as a plain action system, the other one was derived from a UML model by

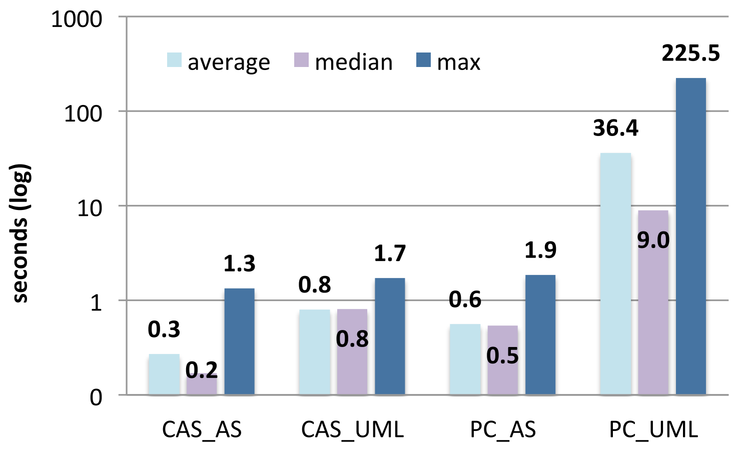


Figure 9.6: Comparison of the average, median, and maximum computation times per mutant for our four models.

MoMuT::UML. Furthermore, we compared two action systems modelling the particle counter. Again, one was a manually modelled plain action system, while the other one was a complex action system automatically generated from a UML model.

The given metrics indicate a significantly higher complexity of the action systems generated from UML models compared to directly modelled action systems. We also checked whether this is directly reflected in the computation times required by our refinement-based test case generation tool. This is obviously the case. However, for the rather simple CAS model, the growth of complexity is not as dramatic as for the particle counter as can be seen from Figure 9.6. It compares the average, the median, and the maximum time required for one mutant of each action system. Note that the Y-axis has a logarithmic scale. The action system derived from UML is still manageable, although the average computation time per mutant has more than doubled. However, for the particle counter, the average time per mutant was raised from less than one second to more than 36 seconds. Considering the maximum values, CAS_AS and CAS_UML do not significantly differ. For PC_AS, the maximum computation time for one mutant was below 2 seconds, but raised to almost 4 minutes for PC_UML. Hence, action systems automatically derived from UML models are more complex and require longer runtimes than manually created plain action systems. We conclude that there is room for improvement in the model transformations implemented in MoMuT::UML's frontend.

In the following, we relate the results for the plain action systems we reported in the previous section with the computation times we achieved before the integration into the MoMuT::UML tool chain (cf. Section 8.2). For the CAS, the computation time was approximately 4 seconds before. After our adaptations to support complex action systems, it is 70 seconds without log file writing. Note that during our adaptions to integrate our tool into MoMuT::UML, we significantly increased the information that is included in the log files. Hence, we compare values that do not contain the time for log file writing. Furthermore, the CAS_AS model used in this chapter directly corresponds to the action system shown in Listing 5.1, while the CAS action system used in our earlier experiments was slightly different regarding the modelling of time. Moreover, the number of mutated action systems slightly increased. In our earlier experiments, we used 207 mutated models. In this section, we considered 255 mutated models of the CAS. This is partly because of the minor changes in the model and mainly due to an additional mutation operator, which inverts Boolean constants. Hence, we compare the average time per mutated model, which increased from 0.02 seconds to 0.27 seconds.

For the particle counter, the runtimes also increased. In our previous experiments, test case generation from the particle counter action system required approximately 3 minutes. With our adapted tool, it increased to 7 minutes without writing logs. Note that for the particle counter, the used action systems are identical. However, due to the additionally introduced mutation operator, the number of mutants raised from 672 to 714. The average time per mutated model increased from 0.27 seconds to 0.59 seconds.

Finally, we have to indicate that these experiments were not conducted on the same computer. However, though a theoretically faster CPU was used now (an Intel i7 with 3.4 GHz instead of an Intel i7 with 2.8 GHz), the runtimes increased by our adaptations to support complex action systems. This cannot be traced back to one single reason, but is a conglomerate of different factors. We now support a richer action system language, which caused us to use a more powerful SMT solver. Instead of SICStus Prolog's built-in constraint solver, we now use Microsoft's Z3, which is an external library and requires additional communication overhead. To conclude, we now support a richer action system language. However, the price to pay is an increased runtime.

10 Combining Refinement and Input-Output Conformance

This chapter is based on joint work with Martin Tappler. Under my supervision, he implemented the ioco check based on my existing code. Furthermore, he performed the experiments and assisted in the analysis of the results.

At the end of Chapter 8, we already pointed out the problem that arises when our refinement relation is used for model-based mutation testing. Our refinement relation is very strict and also leads to non-conformance due to unspecified states in the mutant. However, states are not incorporated in our test cases. Hence, a test case that results from a mutant that differs from the original by an unspecified state, but not by observations, is not able to distinguish the mutated from the original model as we illustrate in the following example.

Example 10.1. Consider an action system that comprises one state variable s, which is an integer between 0 and 2. The initial state is defined as $s = 0$. Furthermore, the action system comprises one observable action a defined as $a :: (true) => (s := 1)$, which is called in the do-od block. The corresponding LTS is depicted on the left-hand side of Figure 10.1. The labels of the LTS states correspond to the valuation of the state variable s. Consider a mutated version of this action system, which redefines action a to be $a :: (true) => (s := 2)$, i.e., the post-state of the action is set to 2 instead of 1. This yields the LTS depicted in the middle of Figure 10.1. According to our refinement relation, the mutant does not refine the original action system due to an unspecified state. The initial state $s = 0$ is unsafe, the trace to the unsafe state is the empty trace. As described in Chapter 8, we construct a test case, which goes one step beyond the unsafe state. In this case, this yields a test case consisting of action a depicted on the right-hand side of Figure 10.1. Obviously, this test case cannot distinguish the original from the mutated system. □

For this example, the generated test case is superfluous as the two systems cannot be distinguished by their traces. However, in other cases we miss mutations because we generate too short test cases. Again, we use an example for illustration.

Example 10.2. We extend the mutated and the original action system from Example 10.1 by an additional action b defined as $b :: (s \neq 1) => (skip)$. The corresponding LTSs are depicted on the left-hand side of Figure 10.2. Our refinement check again identifies the initial state as unsafe. The generated test case is depicted in Figure 10.2 as the second from right LTS. This test case again cannot distinguish the original from the mutant, although the mutant shows unspecified output actions. The test case is too short. A distinguishing test case needs to be one step longer to identify that the mutant allows the unspecified output action b after action a. □

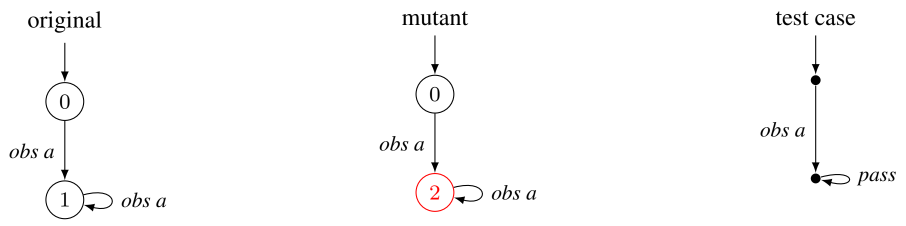


Figure 10.1: The original system cannot be distinguished from the mutant by their visible traces. However, our refinement relation does not hold due to an unspecified state in the mutant. We generate a superfluous test case.

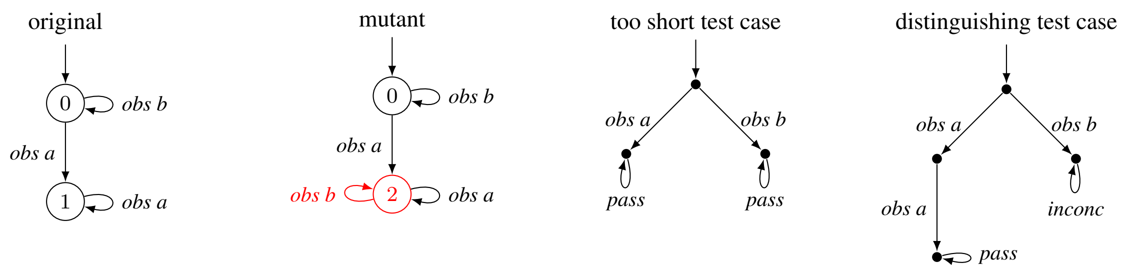


Figure 10.2: Our refinement check results in a too short test case for the two LTSs depicted on the left-hand side.

Additionally to the above described problems with unobservable states, our refinement relation also certifies non-conformance on occurrences of internal actions that are not specified. This may lead to similar problems. In this chapter, we address these problems by additionally considering the Input-Output Conformance (ioco) relation, which is based on visible traces (more precisely the suspension traces, cf. Section 3.2). The ioco relation does not only help in resolving the above described problems. It additionally offers a further advantage as it allows for partial models. It allows an implementation to react arbitrarily to unspecified inputs. Partial-model support is an important feature of model-based approaches as incorporating all aspects of a complex SUT in one monolithic model is hard.

Our idea is to combine our efficient refinement check with a subsequent ioco check, which is more costly in general. We start with a brief description of our implementation of the ioco check and then explain how we combine our refinement relation with ioco. Finally, we report on experimental results.

10.1 Checking for Input-Output Conformance

Our implementation of the ioco check is based on the same concept as used by Ulysses [50, 10]: the synchronous product modulo ioco of the two underlying LTSs is computed on the fly. That is, the LTSs of the mutated and the original action system are explored and immediately checked for ioco conformance. If non-conformance is detected, the exploration of the LTSs is stopped. This on-the-fly approach for ioco checking was first presented by Weiglhofer and Wotawa [200]. The synchronous product modulo ioco of a specification and an implementation is characterised by the following rules:

1. The two systems synchronise on common actions.
2. Inputs that are allowed in the implementation, but not in the specification are not further investigated. They lead to a pass state in the synchronous product modulo ioco. This rule reflects implementation freedom, i.e., implementations may behave arbitrarily after unspecified inputs.
3. The implementation may show fewer outputs than the specification, i.e., an output that is only allowed in the specification, but not by the mutant leads to a pass state in the product LTS.
4. Furthermore, input-enabledness of the implementation is considered by performing the angelic completion of the implementation (cf. Section 3.2), i.e., for inputs in the specification that are not enabled in the implementation, a self-loop labelled by this input is added to the implementation. In this way, the implementation accepts, but ignores the input.
5. Finally, if the implementation shows an output that is not specified, the ioco relation is violated. This leads to a fail state in the synchronous product modulo ioco.

For a formal definition of the synchronous product modulo ioco and a pseudo-code algorithm of its on-the-fly calculation, we refer to Weiglhofer and Wotawa [200].

In our setting, the mutated action system represents the implementation, and the original action system serves as specification. Our implementation of the ioco check explores both action systems in a breadth-first search. It adds δ-loops in quiescent states (Definition 3.16) and performs the τ-closure, i.e., follows internal actions until a visible action is reached. Furthermore, determinisation is applied on the visible actions. This corresponds to the on-the-fly creation of the suspension automaton yielding the suspension traces (Definition 3.17). These suspension traces form the basis for the ioco check (Definition 3.20), i.e., they are used for calculating the synchronous product modulo ioco as described above. Like our non-refinement check, the ioco check is also bounded, i.e., it stops either if non-conformance is identified or if a specified maximum exploration depth is reached. Analogously to our refinement check, we record the trace to the unsafe state with respect to ioco and construct a test case as described in Chapter 8.

As already pointed out in Chapter 8, our exploration of the action systems is based on constraints representing the transition relation. In contrast, Ulysses enumerates all possible parameter valuations and then tests whether these values fulfil the guard of a given action. For large domains, this is inefficient and our constraint-based approach usually performs better – particularly with respect to memory consumption. This was also the main reason why we decided for a re-implementation instead of directly using Ulysses.

10.2 Combination of Refinement and Input-Output Conformance

As already explained above, our refinement relation for action systems is a rather strict conformance relation. It disallows differences of states, which are usually not observable in black-box testing. Furthermore, it is sensitive to τ-actions: if the mutant performs an internal action, this must be specified by the original model. Moreover, refinement does not distinguish between inputs and outputs. Hence, additional inputs in the SUT lead to non-refinement and partial models become useless.

The ioco conformance relation and its variations are better suited for black-box testing, which is illustrated by its application in many tools [37, 126, 112, 68]. It is also used by Ulysses [10], the existing test case generation backend of MoMuT::UML. However, we experienced that a full ioco check of two action systems can be rather costly. This can also be seen from our experimental results, which we present in the next section.

To counteract, we combine our strict, but efficient refinement check with an ioco check. We consider our refinement check, where internal states, internal actions, or unspecified input actions already kill a mutant, as weak mutation testing (Definition 4.15). It determines, whether the necessity condition (Definition 4.12) is fulfilled, i.e., whether a mutation has been reached and has infected the mutated system. To achieve the stronger mutation testing (Definition 4.14), the sufficiency condition (Definition 4.13) has to hold additionally. It states that the observed internal error must propagate to an observable failure. With respect to ioco, it must propagate to an unspecified output action. Hence, we first perform a refinement check. Only if non-refinement is identified, we append an ioco check starting from the unsafe state identified by our refinement check. In this way, the ioco check is more targeted to those parts of the system, which are actually affected by the mutation. This allows for higher exploration depths and hence longer test cases.

Note that our combined approach preserves ioco's support for partial models. If non-refinement is caused by an additional input action in the mutant, it does not propagate in the subsequent ioco check as illustrated below.

Example 10.3. Reconsider the original action system introduced in Example 10.1. Its LTS representation is depicted on the left-hand side of Figure 10.3. The right-hand side shows a mutant that introduces an additional input action x, which is always enabled. This mutant is ioco-conform to the original as the

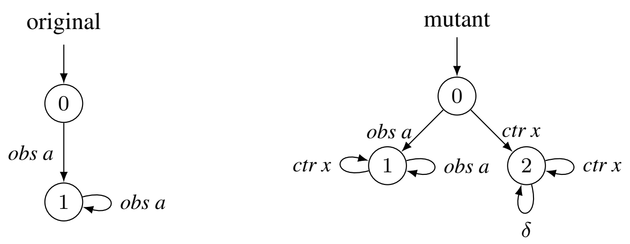


Figure 10.3: Our combined refinement/ioco check allows for partial models.

additional action is an input action and unspecified inputs in the mutant may lead to arbitrary behaviour (cf. Section 3.2). However, the mutant does not refine the original due to the additional action x, because refinement does not distinguish between inputs and outputs. The initial state is unsafe according to our refinement check. In our combined conformance check, this is the starting point for the ioco check, which does not find non-conforming behaviour. Hence, our overall combined conformance check results in conformance. □

Our combined refinement and ioco test case generation is sketched in Algorithm 10.1. As input, it requires the original action system as, its initial state $init$, and a corresponding set of mutated action systems $mutants$. Furthermore, the maximum depths for the refinement check ($maxRef$) and the ioco check ($maxIoco$) need to be specified by the user. As our most optimised refinement checker (cf. Chapter 7) is used, we pre-compute the state space up to the given depth for the refinement check in Line 1. Then, we iterate over the mutants (Line 2). Each mutant is checked for refinement in Line 3. If non-refinement is detected, the function $checkRefinement$ returns the unsafe state u together with a trace leading to this state ($tr2UnsafeRef$). If the mutant asm refines the original as up to the given depth, nil is returned for the unsafe state and we move on to the next mutant. If an unsafe state u has been found (Line 4), we perform an ioco check starting at the unsafe state u with a maximum depth specified by the user. The function $checkIoco$ returns a trace that leads from the unsafe state u, which was determined by the refinement check, to the unsafe state of the ioco check. If the mutant is ioco-conform to the original (up to the given depth $maxIoco$), the returned trace is nil. In this case, no test case is generated and we move on to the next mutant. If the trace to the unsafe state of the ioco check is unequal to nil (Line 6), it is appended to the trace of the refinement check forming the overall trace to the unsafe state (Line 7). This trace starts at the initial state of the action system and leads to the unsafe state, which triggers an ioco difference. It serves as the basis for a test case (Line 8). The function $constructSaveTc$ performs the test case construction (cf. Chapter 8). It takes the complete trace to the unsafe state, the original action system, and its initial state. The resulting test case is saved in a file. Note that the resulting test cases may have a length of up to $maxRef + maxIoco$, i.e., the sum of the maximum depths of the two conformance checks.

10.2.1 Under-Approximation

This combination of our refinement check with a subsequent ioco check results in a notion of conformance that is slightly weaker than ioco. Our refinement check classifies certain mutants as refining, although they are not ioco-conform. In this way, also our combination of refinement/ioco classifies such mutants as conforming and does not generate a test case, where a pure ioco check would result in a test case. Therefore, our combined approach results in an under-approximation of the test suite that would be generated by a pure ioco check given the same set of mutants. On the other hand, our combined approach has also advantages. It is significantly faster than an ioco check as will be seen later from our experimental results. In the following, we discuss the three cases where non-conformance with respect to ioco is not detected by our combined approach.

Algorithm 10.1 $\mathit{combinedRefIocoTcg}(\mathit{as}, \mathit{init}, \mathit{mutants}, \mathit{maxRef}, \mathit{maxIoco})$

```
states := findAllStates(as, init, maxRef)
for all asm ∈ mutants do
   (u, tr2unsafeRef) := checkRefinement(states, as, asm)
   if u ≠ nil then
      tr2UnsafeIoco := checkIoco(as, asm, u, maxIoco)
      if (tr2UnsafeIoco ≠ nil) then
         tr2Unsafe := tr2UnsafeRef⌢tr2UnsafeIoco
         constructSaveTc(as, init, tr2Unsafe)
      end if
   end if
end for
```

Quiescence

In our refinement relation, we neglect quiescence (Definition 3.16) as it is not encoded in our predicative semantics for action systems. For the ioco check, it is added during the exploration of the underlying LTSs similarly as explained in Chapter 8. As a consequence, an implementation may conform to a specification with respect to our refinement relation, but not with respect to ioco. In this way, our combined refinement/ioco check classifies certain mutants as conforming, although ioco does not hold. Consider the following example.

Example 10.4. Consider the three action system snippets depicted in Figure 10.4. Assume that the state of each action system consists of one variable s, which is an integer between 0 and 2. The first two lines of each listing define two actions a and b. Their composition via non-deterministic choice in the do-od block is shown in the last line. Furthermore, assume that the initial value for the state variable s is 0 in each action system. The listing at the left-hand side represents the original model, while the other two action systems are mutations thereof. In fact, both mutants disable action b. Mutant 1 in the middle establishes this by setting b's guard to false. Mutant 2 on the right-hand side makes the only state that satisfies b's guard, i.e., $s = 1$, unreachable by changing the post-state of action a from 1 to 2. The LTSs representing these action systems are depicted in Figure 10.5.

Our refinement relation does not consider quiescence, i.e., the δ-loops highlighted in blue in Figure 10.5 have no influence on refinement and mutant 1 refines the original as it does neither reach unspecified states nor shows unspecified actions. Although mutant 1 refines the original, it is not ioco-conform to the original as $\{\delta\} = \mathit{out}(\text{mutant 1 } \mathit{after}\ \langle a\rangle) \not\subseteq \mathit{out}(\text{original } \mathit{after}\ \langle a\rangle) = \{b\}$ (cf. Definition 3.20). Since refinement holds, our combined refinement and ioco check also considers mutant 1 as conforming and no test case will be generated in contrast to a pure ioco check.

Regarding ioco, the situation with mutant 2 is completely the same as with mutant 1. However, for refinement mutant 1 and mutant 2 are different. While mutant 1 refines the original, mutant 2 does not refine the original as it reaches an unspecified state by action a (2 instead of 1). □

This example demonstrates that the ignorance of quiescence causes our refinement check and as a consequence our combination of refinement and ioco to classify certain mutants as conforming, although they are non-conforming with respect to ioco. In this way, our combined refinement/ioco check generates fewer test cases than a stand-alone ioco check. Note that this difference is restricted to cases where quiescence is not caused by an unspecified state in the mutant.

original	mutant 1	mutant 2
`a::(s = 0) => (s := 1)`	`a::(s = 0) => (s := 1)`	`a::(s = 0) => (s := 2)`
`b::(s = 1) => (s := 2)`	`b::(false) => (s := 2)`	`b::(s = 1) => (s := 2)`
`a [] b`	`a [] b`	`a [] b`

Figure 10.4: Code snippets of an original action system and two possible mutants.

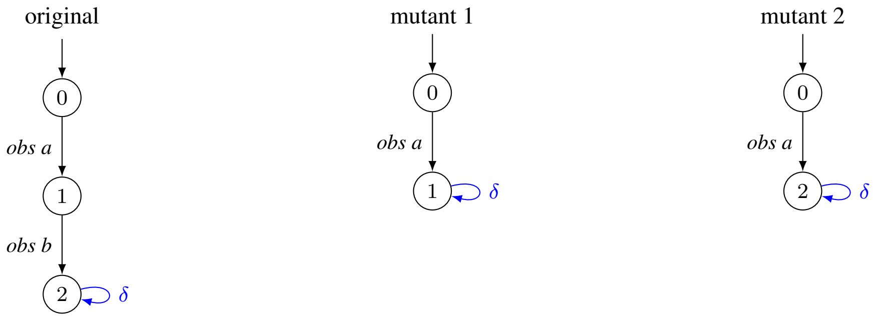


Figure 10.5: The LTSs representing the action systems shown in Figure 10.4.

Input-Enabledness

Furthermore, the differences between our combined refinement/ioco check and a pure ioco check are partly caused by the ioco-assumption that implementations are input-enabled (Definition 3.19). For ioco, implementation models are implicitly made input-enabled by angelic completion, i.e., by adding self-loop transitions labelled by inputs that are not enabled in a state (cf. Section 3.2). In this way, unknown inputs are always accepted, but ignored by implementations. Again, we use an example for illustration.

Example 10.5. Consider the LTSs depicted in Figure 10.6. The left-hand side represents the original model, while the right-hand side is a mutant created by setting the guard of the input action x to false. The mutant refines the original model as the blue loops labelled by *ctr x* in the mutant are not part of the mutant originally and not considered for our refinement check. They have only been added for the ioco check to fulfil ioco's requirement that the implementation, in our case the mutant, has to be input-enabled. This input-enabledness causes the mutant to be not ioco-conform to the original model: $\{a\} = out(\text{mutant } after\ \langle x\rangle) \not\subseteq out(\text{original } after\ \langle x\rangle) = \{b\}$ (cf. Definition 3.20). For this example, our refinement check and as a consequence our combination of refinement and ioco does not generate a test case, for which the pure ioco check results in a distinguishing test case. □

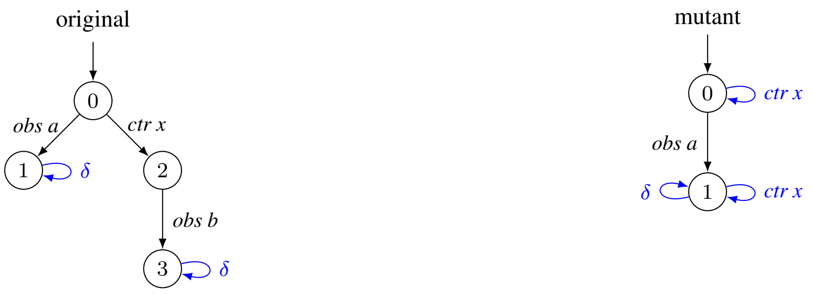


Figure 10.6: Two LTSs used for demonstration of differences between a pure ioco check and our combined refinement/ioco check, which are caused by the input-enabledness assumption of ioco.

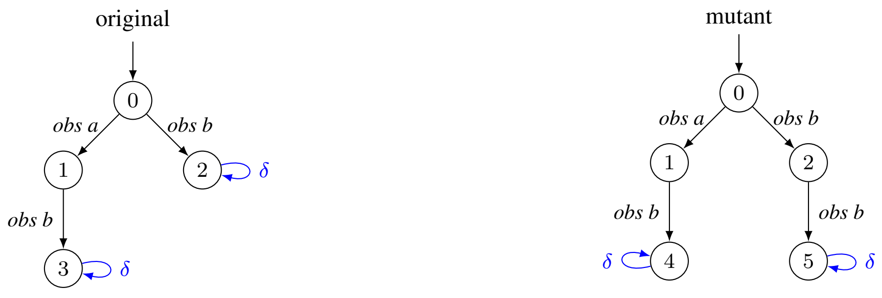


Figure 10.7: These LTSs are used to demonstrate that our combination of refinement and ioco misses a test case due to unimplemented backtracking.

Stop at First Unsafe State

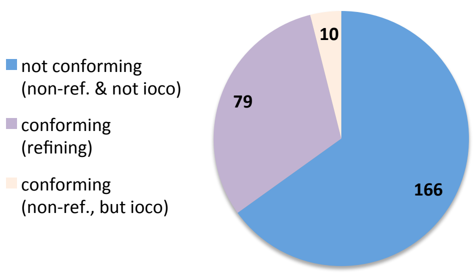
Finally, our refinement/ioco check might wrongly classify mutants as conforming due to the stopping criterion of our refinement check. Our refinement check searches for only one unsafe state with respect to refinement and then performs an ioco check starting at this state. If this ioco check does not find violations of ioco, then the mutant is classified to be conforming. However, it is possible that there exist further unsafe states with respect to refinement that actually propagate to an ioco difference. Hence, to avoid this problem we would have to backtrack, search for another unsafe state with respect to refinement and check if this one propagates. This iterative process stops either if a violation of ioco can be identified, or if no further unsafe states can be found by our refinement checker.

Example 10.6. Consider the LTSs depicted in Figure 10.7. Clearly, the mutant is not ioco to the original since $\{b\} = \mathit{out}(\text{mutant } \mathit{after}\ \langle b \rangle) \not\subseteq \mathit{out}(\text{original } \mathit{after}\ \langle b \rangle) = \{\delta\}$ (cf. Definition 3.20). However, it is possible that our implementation combining refinement and ioco classifies the mutant as conforming. In our notion of refinement, state 1 is an unsafe state due to unspecified successor states: from state 1, the mutant reaches state 4 by action b, while the original only specifies state 3 after action b. Hence, we start an ioco check in state 1, which results in conformance. Currently, our implementation stops here and hence, misses the ioco violation in the other branch. If we had implemented a backtracking facility that returns to the refinement check, a second unsafe state would be found at state 2 and the subsequent ioco check beginning in state 2 would immediately reveal the non-conformance. □

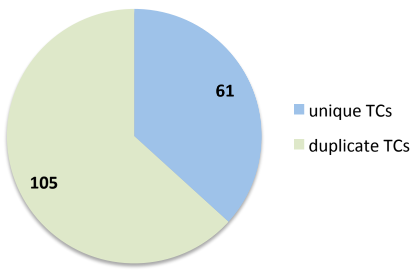
As described above, our combination of refinement and ioco may wrongly classify mutants to be conforming, although they are not ioco to the original. Hence, our combined approach results in an under-approximation of the test suite that would be generated by a pure ioco check given the same set of mutants. However, our approach seems to be a good trade-off between computation time and fault coverage, as will be seen from our experimental results that we describe in the following.

10.3 Experimental Results

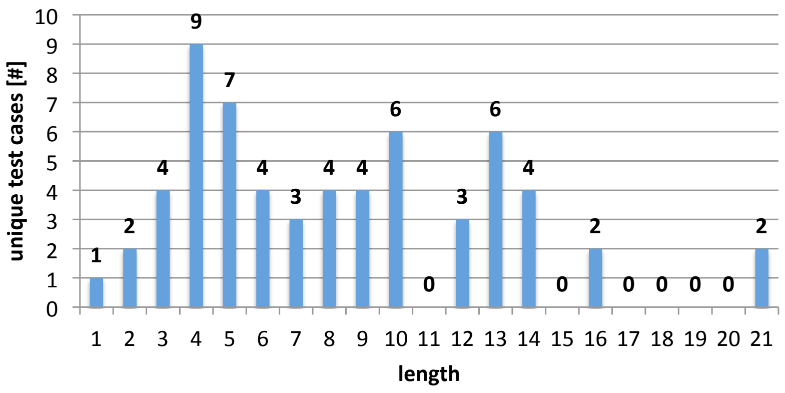
To evaluate our combined refinement/ioco checker implementation, we applied it to the four models already used with our pure refinement checker (Section 9.3). The model CAS_AS specifies the CAS directly as a plain action system, while CAS_UML is a complex action system that models exactly the same behaviour, but has been derived by MoMuT::UML from a UML model. Similarly, PC_AS models the particle counter control logic as a plain action system and PC_UML is a complex action system derived from a UML model specifying the same behaviour as PC_AS. We use the same mutated models as in Section 9.3. We also conducted a stand-alone ioco check for these models and compare it to our combined refinement/ioco check. All experiments were performed on the same computer as



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.8: Test case generation with our combined refinement and ioco check for CAS_AS.

in Section 9.3. It runs a 64-bit Linux (Ubuntu 12.04), is equipped with 8 GB RAM, and an Intel i7 quad-core processor (3.4 GHz).

10.3.1 Car Alarm System

Plain Action System

For CAS_AS, we set the exploration depth limit of the refinement check to 20 and we allowed for further 20 steps for the ioco check. As the model shows new states only up to depth 11 (cf. Table 9.1), this ensures that the whole model is covered.

Figure 10.8 gives an overview of the test case generation results from our combined refinement/ioco check. Figure 10.8a divides the given model mutants into conforming and non-conforming mutants. In total, we used 255 mutated models. Thereof, 79 (31%) conform to the original as they refine the original. Another 10 mutants (4%) are conforming although they do not refine the original, because the non-refinement did not propagate to an ioco difference. Hence, in most cases non-refinement propagates to an ioco violation. In total, we have 89 conforming mutants (35%). The remaining 166 model mutants (65%) do not conform to the original, i.e., they do not refine the original and the non-refinement propagates to an ioco violation. For each of these non-conforming mutants, a test case has been generated. As already pointed out in Chapter 8, many test cases are duplicates in our current setting. Figure 10.8b illustrates that 105 test cases (63%) are duplicates of others, and 61 unique test cases remain after the removal of these duplicates. The lengths of the unique test cases are depicted in Figure 10.8c. The longest test cases comprise 21 consecutive actions, although the model shows only new states up to depth 11. This is due

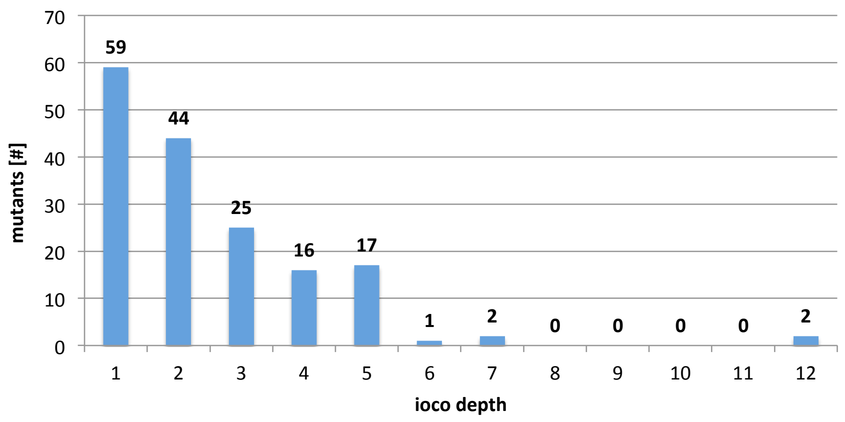


Figure 10.9: Overview of the ioco depths in our combined refinement/ioco check for CAS_AS.

to our combination of refinement and ioco. If non-refinement is identified deep in the model and the ioco check also requires many steps, this leads to longer test cases. For example, one test case had an unsafe state for refinement at depth 9 and the ioco check needed 12 steps to find an unsafe state. In this way, the overall test case has a length of 21 steps.

The depths needed for propagation of non-refinement to an ioco difference are depicted in Figure 10.9. As can be seen from the diagram, most non-refinement issues propagate to an ioco violation within 5 steps. For 59 mutants, non-refinement immediately leads to an ioco difference (depth 1). This is the case if non-refinement was caused by an unspecified output action, which coincides with an ioco violation. Only 1 model mutant required an ioco depth of 6 steps and 2 mutants required 7 steps. However, there are also 2 mutants that require 12 steps for the ioco check. Note that the depths of the unsafe states found by the refinement check coincide with those reported in the previous chapter, where we reported on the stand-alone refinement check (Section 9.4.1).

Table 10.1 gives detailed information about the computation times required for our combined refinement and ioco check. It splits the total computation time into those parts spent on conforming and not conforming mutants, which are restated in the first row of the table. For conforming mutants, it furthermore distinguishes between those that refine the original and those that are non-refining, but did not propagate. The table gives computation times for all mutants considered in a column (Σ), for the average per mutant (ϕ), and the maximum time per mutant (max). We state runtimes for the following tasks:

1. The time required for finding all reachable states. Note that this is performed once for all mutants and cannot be split between conforming and non-conforming mutants.
2. The time required for the refinement checks.
3. The time for the ioco checks.
4. The time used for test case construction, i.e., the time to make a test case out of the traces to the unsafe states. It includes the time for writing the test cases into files on the hard drive.
5. Finally, we state the total computation time. It comprises the time for both conformance checks and the time used for test case construction. However, it does not include the time for finding the reachable states as this is done once and is reused for all mutants. Hence, it cannot be split between conforming and non-conforming mutants. Furthermore, we use two categories for the total computation time: one with activated logs and one without log files. As already discussed in the previous chapter, our tool writes comprehensive log files, which can be used for debugging and our evaluations. However, this is a considerable overhead for such small examples and is not required for test case generation in practice, where it can be deactivated.

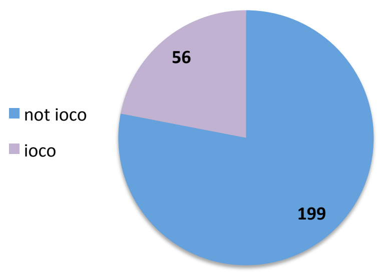
		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		79	10	166	255
time – find states	Σ	-	-	-	0.2
time – ref. check	Σ	0.43	0.04	0.94	1.41
	ϕ	0.01	0.00	0.01	0.01
	max	0.02	0.01	0.02	0.02
time – ioco check	Σ	-	0.89	13.58	14.47
	ϕ	-	0.09	0.08	0.06
	max	-	0.10	0.57	0.57
time – tc constr.	Σ	-	-	71.2	71.2
	ϕ	-	-	0.43	0.28
	max	-	-	1.64	1.64
total computation time	Σ	3.1 min	27.3	8 min	11.6 min
	ϕ	2.39	2.73	2.91	2.74
	max	3.14	4.10	4.18	4.18
total computation time without log	Σ	0.45	0.93	1.4 min	1.5 min
	ϕ	0.01	0.09	0.52	0.34
	max	0.02	0.10	1.75	1.75

Table 10.1: Test case generation via the combined refinement/ioco check for the CAS_AS model. The maximum search depth for the refinement check was set to 20, the subsequent ioco check was also limited by a depth of 20, i.e., the overall maximal search limit was 40 steps. All values are given in seconds unless otherwise noted.

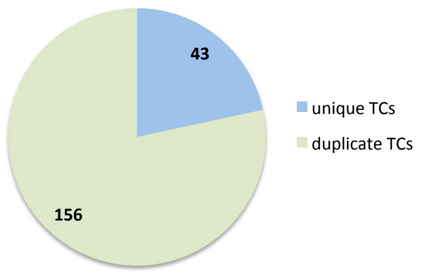
We will not discuss all values in Table 10.1 in detail. Instead we highlight the most interesting numbers. First of all, the refinement check uses 1.4 seconds of the overall computation time, while the ioco check requires 14.5 seconds. Furthermore, for this simple model, the test case construction takes longer than both conformance checks together (71 seconds vs. 16 seconds). However, the majority of the overall runtime is used for our extensive logs as can be seen from the two sections at the bottom of the table. The overall computation time including logs is almost 12 minutes. By exclusion of the time required for writing the logs, it decreases to 1.5 minutes.

For comparison, we also performed a pure ioco check using our implementation described in Section 10.1. We specified a maximum depth of 20, which is sufficient to fully cover this model. In this way, we have a comparable setting to our combined conformance check, where we used 20+20 steps for the search depths. Figure 10.10 summarises the results. Out of the 255 mutated models, 56 are ioco-conform to the original (22%) and the remaining 199 mutants are non-conforming (cf. Figure 10.10a). For each of these mutants, a test case has been generated. Again, there were many duplicates among the test cases (156, i.e., 78%), which indicates that there is a certain redundancy in the used model mutations. Hence, we obtain a test suite consisting of 43 unique test cases (Figure 10.10b). These test cases have a maximum length of 16 consecutive actions (cf. Figure 10.10c).

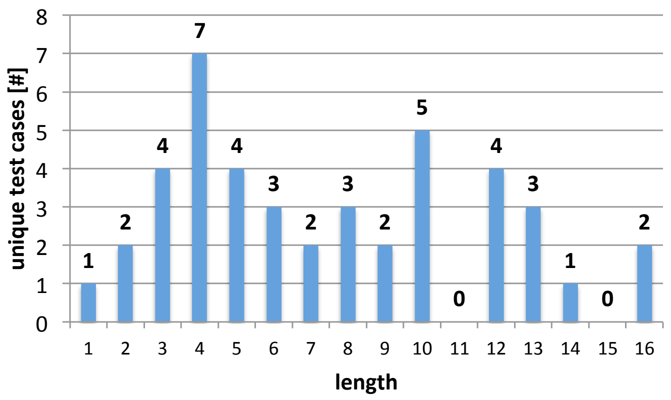
Comparing the diagrams for our combined conformance check (Figure 10.8) and this pure ioco check (Figure 10.10) leads to the following observations. The ioco check classifies 56 mutants to be conform to the original, while our combined conformance check results in 89 cases in conformance. Hence, our combined conformance check wrongly classifies 33 model mutants as conforming. This is not a surprise. We already explained possible reasons in Section 10.2.1. We analysed all 33 concerned mutants and found out that the reason was either that our refinement check disregards quiescence or that it does not consider input-enabledness of the mutant. We used Ulysses to execute the 61 test cases generated by our combined conformance check on the 33 affected model mutants. Each model mutant was killed, i.e., our



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.10: Test case generation based on a stand-alone ioco check for CAS_AS.

generated test suite does effectively achieve the same coverage of model mutants as the full test suite generated by the ioco check. However, the test suite generated by our combined check is larger than the ioco test suite – both in terms of the number of test cases as well as in the length of the generated test cases. Our combined approach yields 61 unique test cases with a maximum length of 21. The ioco check results in 42 unique test cases with a maximum length of 16. This can be explained by the fact that the ioco check in our combined approach is only applied on sub-parts of the model, which are specified by the refinement check. In contrast, the ioco check starting from the initial state investigates the whole state space and has more possibilities to identify an ioco violation. It may find ioco differences in lower depths, which are not considered by the combined approach if non-refinement was found in another part of the model.

Table 10.2 summarises the computation times for the ioco check. It is structured analogously to Table 10.1. Again, we will not discuss the whole table, but highlight interesting facts. As in our combined refinement and ioco check, the test case construction needs more time than the conformance check itself (78 vs. 19 seconds). Furthermore, the log files cost a considerable amount of time (73% of the overall time). The ioco check and our combined conformance check, take almost the same amount of time (given that we do not produce log files). The ioco check required 1.6 minutes, while our combination needed 1.5 minutes (cf. Table 10.1). Hence, for this small model, we cannot achieve a performance gain by combining refinement and ioco checking. However, compared to Ulysses, our implementation is faster. Ulysses was used in a comparable configuration, i.e., we used the same search depth of 20, stop at the first unsafe state, etc. While we need 1.5 minutes with both of our approaches, Ulysses runs approximately 6.5 minutes. Hence, our re-implementation of the ioco check is faster for this example. However, this improvement is not crucial for the small CAS example. In the following, we discuss the

		not ioco	ioco	total
mutants [#]		199	56	255
time – ioco check	Σ	16.21	2.68	18.89
	ϕ	0.08	0.05	0.07
	max	0.91	0.09	0.91
time – tc constr.	Σ	78.16	-	78.16
	ϕ	0.39	-	0.31
	max	1.17	-	1.17
total computation time	Σ	4.9 min	59.63	5.9 min
	ϕ	1.49	1.06	1.40
	max	3.20	1.40	3.20
total computation time without log	Σ	1.6 min	2.68	1.6 min
	ϕ	0.47	0.05	0.38
	max	1.57	0.09	1.57

Table 10.2: Test case generation via ioco checking up to depth 20 for the CAS_AS model. All values are given in seconds unless otherwise noted.

results achieved with the complex action system of the CAS generated by MoMuT::UML.

Complex Action System from UML Model

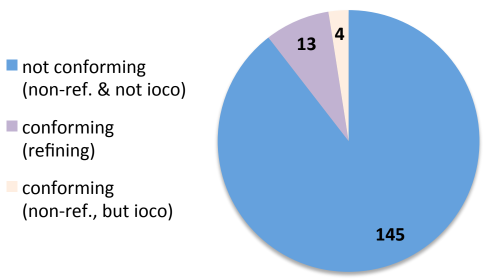
For the CAS_UML model, we conducted the same experiments as for CAS_AS. Again, we used 20 as the maximum exploration depth for both the refinement and the ioco check.

Figure 10.11 gives an overview of the results from our combined refinement/ioco check. Figure 10.11a illustrates that the majority of all model mutants was non-conforming (90%). For the other 10%, conformance was either caused by refinement (13 mutants) or by the fact that non-refinement did not induce an ioco difference (4 mutants). As for CAS_AS, non-refinement propagated to an ioco violation in most cases. Thereby, 145 test cases were generated, whereof 115 are duplicates of others (79%). The final test suite consists of 30 unique test cases (Figure 10.11b). Figure 10.11c shows the lengths of these 30 test cases. The longest test case consists of 13 consecutive actions.

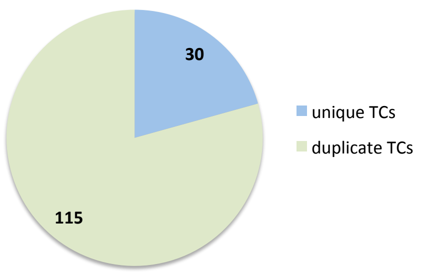
For the non-conforming mutants (non-refining and not ioco-conform), Figure 10.12 illustrates the depths required by the ioco check, i.e., the depths required for the propagation of non-refinement to an observable failure with respect to ioco. Like for the CAS_AS model (cf. Figure 10.9), the required depths are below 5 steps for the majority of the mutations. Only 7 mutations required a depth of 8 and 9 respectively.

Table 10.3 summarises the computation times of our combined refinement and ioco check. It has the same structure as Table 10.1. Again, the time used by the refinement checks is shorter than the time required for the ioco checks (1 vs. 2.2 minutes). The time used by test case construction is smaller than the time for the conformance checks (1.3 minutes vs. 3.2 minutes). Again, our log files cause a substantial amount of the runtime (59%). Overall, the test case generation takes 4.6 minutes if no log files are produced.

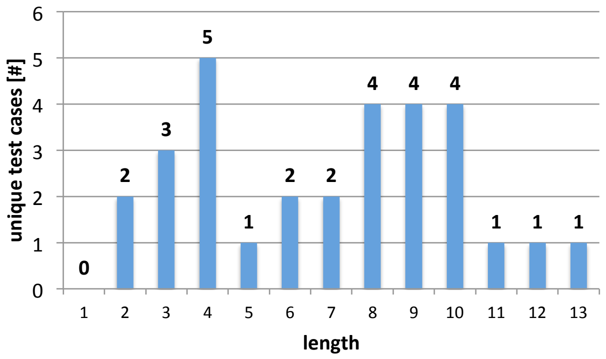
The results of our ioco check for the CAS_UML model with a maximum search depth of 20 are illustrated in Figure 10.13. Figure 10.13a shows that 17 mutants were ioco-conform to the original model (10%). Each of the remaining 145 mutants produced a test case. Thereof, only 27 test cases (19%) remained after removing the duplicates (Figure 10.13b). The unique test cases show lengths up to 13 consecutive steps (Figure 10.13c). Compared to our combined refinement/ioco check, the generated test cases have the same maximum lengths. Also, the number of test cases is almost equal: 27 test cases



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.11: Test case generation with our combined refinement and ioco check for CAS_UML.

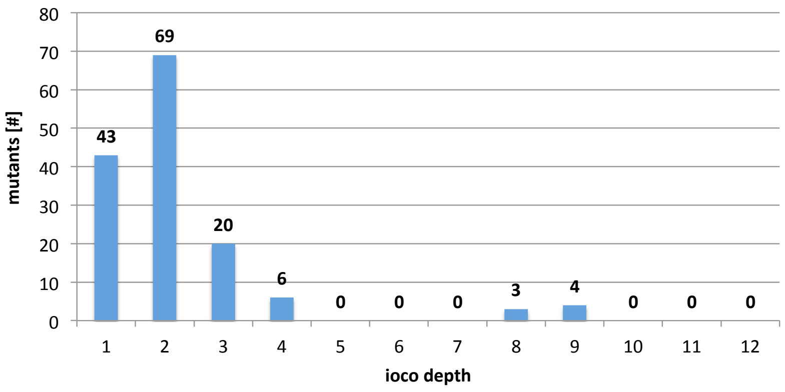


Figure 10.12: Overview of the ioco depths for CAS_UML.

were generated by the ioco check and 30 test cases by our combined conformance check. Furthermore, both conformance checks resulted in the same set of conforming mutants, i.e., for this model and the given model mutants, we did not under-approximate, but correctly identified all non-conforming mutants. We took a deeper look into the model and some mutations and found out that this model assigns many intermediate results in state variables. Thereby, most mutations involve a difference in the states of the mutated models, which is detected by our refinement check.

Table 10.4 gives information on the computation times required by the ioco check. It has the same structure as Table 10.2 describing the runtimes for the CAS_AS model. The time required for ioco checking amounts to 3.7 minutes, while the time for test case generation is approximately 1.4 minutes.

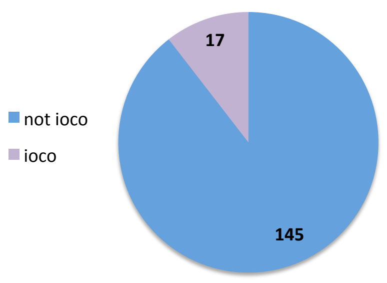
		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		13	4	145	162
time – find states	Σ	-	-	-	3
time – ref. check	Σ	4.03	1.63	56.41	62.07
	ϕ	0.31	0.41	0.39	0.38
	max	0.41	0.44	0.53	0.53
time – ioco check	Σ	-	17.71	1.9 min	2.2 min
	ϕ	-	4.43	0.79	0.81
	max	-	4.48	2.01	4.48
time – tc constr.	Σ	-	-	1.3 min	1.3 min
	ϕ	-	-	0.55	0.49
	max	-	-	1.48	1.48
total computation time	Σ	35.97	28.54	10.1 min	11.2 min
	ϕ	2.77	7.14	4.20	4.16
	max	3.29	7.23	7.53	7.53
total computation time without log	Σ	4.25	19.4	4.2 min	4.6 min
	ϕ	0.33	4.85	1.74	1.7
	max	0.43	4.89	2.77	4.89

Table 10.3: Test case generation via the combined refinement/ioco check for the CAS_UML model. The maximum search depth for the refinement check was set to 20, the subsequent ioco check was also limited by a depth of 20, i.e., the overall maximal search limit was 40 steps. All values are given in seconds unless otherwise noted.

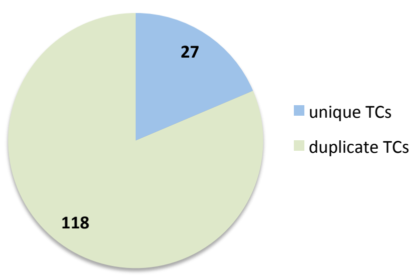
		not ioco	ioco	total
mutants [#]		145	17	162
time – ioco check	Σ	2.7 min	59.97	3.7min
	ϕ	1.11	3.53	1.36
	max	2.52	4.84	4.84
time – tc constr.	Σ	83.21	-	83.21
	ϕ	0.57	-	0.51
	max	1.47	-	1.47
total computation time	Σ	6.5 min	78.31	7.8 min
	ϕ	2.69	4.61	2.89
	max	4.53	6.21	6.21
total computation time without log	Σ	4.1 min	59.97	5.1 min
	ϕ	1.69	3.53	1.88
	max	3.54	4.84	4.84

Table 10.4: Test case generation via ioco checking up to depth 20 for the CAS_UML model. All values are given in seconds unless otherwise noted.

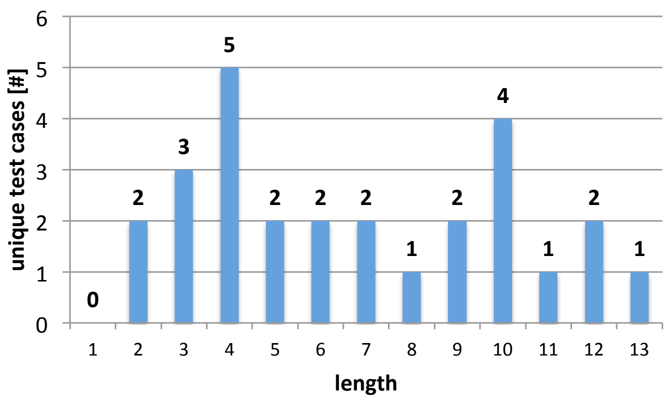
The overall computation time including the time consumed by producing log files is almost 8 minutes. For this model, the time required by writing statistics into log files is not the main part of the overall time. Without log files, the runtime reduces to 5 minutes. What is also worth mentioning, is that the average time required by the ioco check for a conforming mutant is higher than the average time for a non-conforming mutant (3.5 vs. 1.1 seconds). The fact that equivalent mutants raise the overall computation time for ioco checking has already been observed with Ulysses [10].



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.13: Test case generation based on a stand-alone ioco check for CAS_UML.

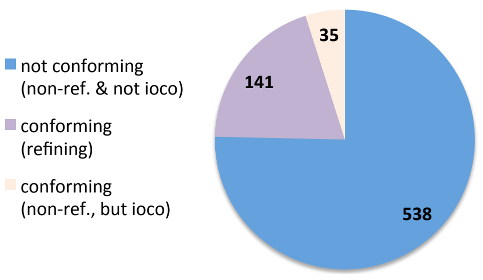
Again, we compare the runtime achieved with our ioco-based test case generator to the computation time required by Ulysses. Given the same search depth of 20, Ulysses needs approximately 10 minutes, which is acceptable. However, our re-implementation is more efficient: it takes only 5 minutes. In the next section, we will see that our efficiency in the ioco checker implementation is crucial for more complex systems like the particle counter.

10.3.2 Particle Counter

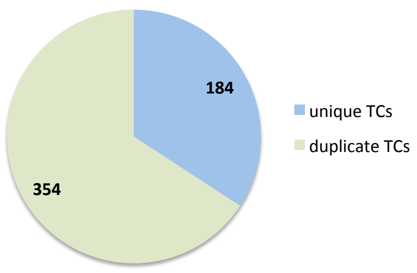
Plain Action System

The results of our combined refinement/ioco check for the PC_AS model are depicted in Figure 10.14. We used a maximum depth of 30 for the refinement check and 20 for the subsequent ioco check. Note that the depth required to find all states of this model is 28. As can be seen from Figure 10.14a, 538 (75%) of the 714 model mutants are non-conforming. The remaining 25% split into refining mutants (141 mutants) and non-refining mutants, where the non-refinement did not propagate to an ioco difference (35 mutants). From the 538 generated test cases, 354 are duplicates of others (66%). After their removal, the resulting test suite consists of 184 unique test cases (Figure 10.14b). The lengths of these test cases is illustrated in Figure 10.14c. The longest test case consists of 22 consecutive actions. There are only a few very short test cases (only 4 test cases up to depth 4). The most common length is 12 (29 test cases).

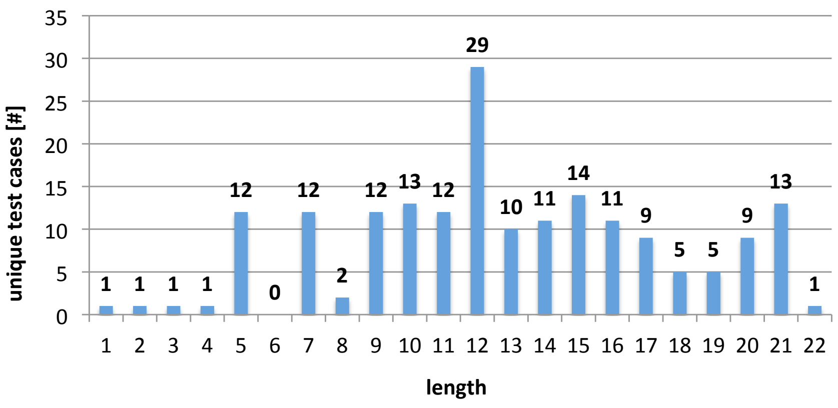
For those non-refining mutants where a propagation to an ioco difference was identified, Figure 10.15 summarises the depths required by the ioco check to find a difference. Like for our previous models, in most cases a maximum depth of 5 is sufficient. Only two mutants required higher depths (7 and 10).



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.14: Test case generation with our combined refinement and ioco check for PC_AS.

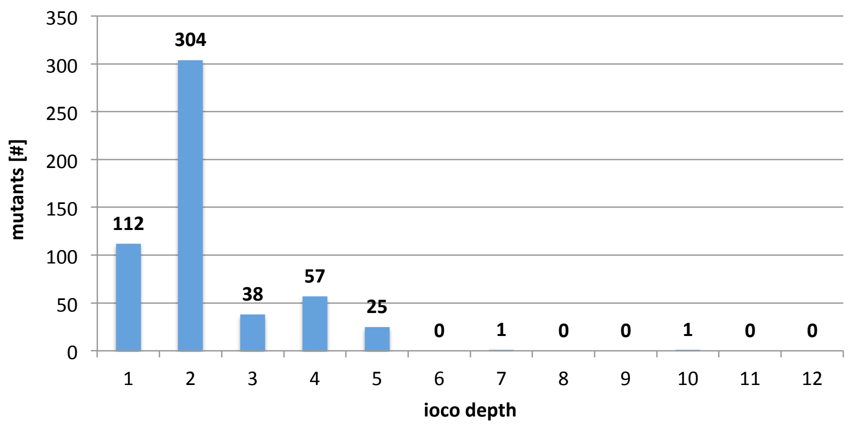
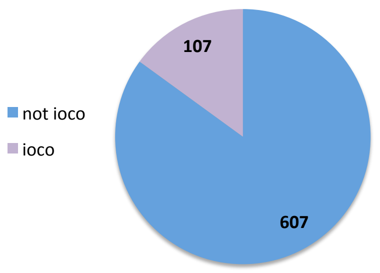


Figure 10.15: Overview of the ioco depths in our combined refinement/ioco check for PC_AS.

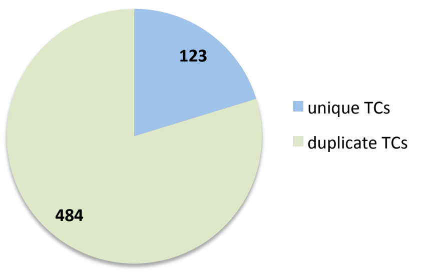
Table 10.5 reports on the required computation times. It is structured analogously to the tables presented earlier in this chapter. The only difference is that we added values for the quartiles where appropriate ($Q_1/Q_2/Q_3$). For this model, the analysis of the runtimes of the individual tasks gives interesting insights. First of all, the overall time for the refinement checks up to depth 30 amounts to less than 3 minutes. In contrast, the ioco checks, which are bounded by a depth of 20, almost require 5 hours. By trend, equivalent mutants cause longer runtimes of our ioco checker. As can be seen from the table, the average time for the ioco check of a mutant that is non-refining, but afterwards ioco is 7.9 minutes. For a mutant, where non-refinement propagates to an ioco violation, the average time for the

		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		141	35	538	714
time – find states	Σ	-	-	-	12.3
time – ref. check	Σ	1.7 min	0.58	1.1 min	2.8 min
	ϕ	0.73	0.02	0.12	0.24
	$Q_1/Q_2/Q_3$	0.8 / 0.9 /1	0.01 / 0.01 / 0.01	0.01 / 0.04 / 0.2	0.01 / 0.05 / 0.3
	max	1.39	0.24	0.75	1.39
time – ioco check	Σ	-	4.6 h	6.5 min	4.7 h
	ϕ	-	7.9 min	0.72	23.76
	$Q_1/Q_2/Q_3$	-	1.7 min / 2.7 min / 3.2 min	0.2 / 0.3 / 0.8	0.1 / 0.2 / 0.8
	max	-	3.2 h	18.3	3.2 h
time – tc constr.	Σ	-	-	6 min	6 min
	ϕ	-	-	0.67	0.5
	max	-	-	1.98	1.98
total computation time	Σ	7.3 min	4.6 h	35.1 min	5.3 h
	ϕ	3.1	7.9 min	3.9	27
	$Q_1/Q_2/Q_3$	3 / 3.2 / 3.4	1.7 min / 2.8 min / 3.2 min	3.1 / 3.6 / 4.3	3 / 3.5 / 4.2
	max	4	3.2 h	22	3.2 h
total computation time without log	Σ	1.7 min	4.6 h	13.6 min	4.9 h
	ϕ	0.73	7.9 min	1.5	24.5
	$Q_1/Q_2/Q_3$	0.8 / 0.9 / 1	102 / 163 / 192	0.6 / 1.2 / 1.8	0.6 / 1.1 / 1.8
	max	1.4	3.2 h	19.7	3.2 h

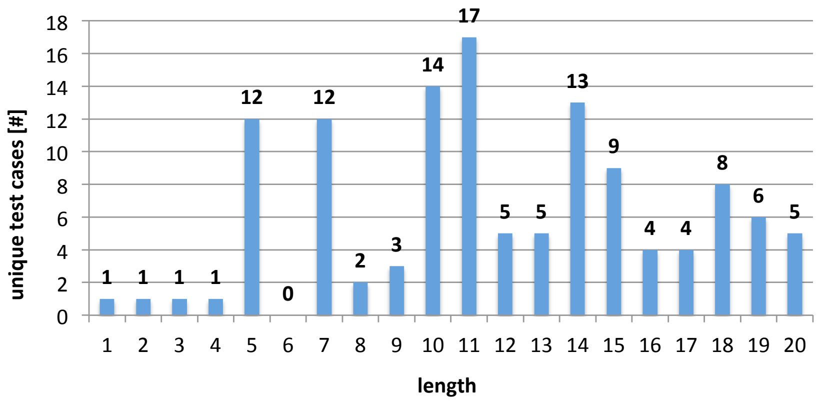
Table 10.5: Test case generation via the combined refinement/ioco check for the PC_AS model. The maximum search depth for the refinement check was set to 30, the subsequent ioco check was limited by a depth of 20, i.e., the overall maximal search limit was 50 steps. All values are given in seconds unless otherwise noted.



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.16: Test case generation based on a stand-alone ioco check for PC_AS.

ioco check is only 0.7 minutes. The high average for the ioco-conforming mutants seems to be caused by a particular outlier, which required 3.2 hours for its ioco check. Overall, the computation time for our combined conformance check is approximately 5 hours. Given these high runtimes caused by the ioco check, our log files do not cause a substantial overhead.

Again, we also performed a stand-alone ioco check. To achieve computation times in the same order of magnitude, we decided not to use the same depth as for the refinement check for this more complex model. While the refinement check was limited to a depth of 30, we restricted the maximum depth for the ioco check to 20, which entails that we cannot fully explore the whole model and in turn possibly cannot identify some non-conforming mutations. Figure 10.16 summarises the results. Approximately 15% of the model mutants are ioco-conform up to depth 20. The remaining 85% of the model mutants resulted in 607 test cases. Thereof, 80% were duplicates. The final test suite consists of 123 unique test cases (cf. Figure 10.16b). Their lengths are depicted in Figure 10.16c. The longest test cases consists of 20 consecutive steps, which is the maximal possible length due to our depth restriction. Similarly as our combined check, the ioco check produced only 4 short test cases with a length of up to 4 steps.

In comparison to our combined refinement and ioco check, the ioco check produces a smaller test suite (123 test cases with a maximum length of 20 vs. 184 test cases with a maximum length of 22). We furthermore compared the sets of conforming and non-conforming mutants of both approaches. We found out that the ioco check finds 10 mutants to be ioco-conform, while they are not in our combined check. This is due to the limited depth of the ioco check. Hence, a depth of 20 is not sufficient for this model and the given mutants. In turn, our combined conformance check cannot kill 79 model mutants that are identified as non-conforming by the ioco check. This is due to the differences between refinement and ioco, which we described in Section 10.2.1. However, we executed the test suite produced by our

		not ioco	ioco	total
mutants [#]		607	107	714
time – ioco check	Σ	1.7 h	5.2 h	6.9 h
	ϕ	10.1	2.9 min	35
	$Q_1/Q_2/Q_3$	0.2 / 1.9 / 12	70 / 71 / 79	0.2 / 3.4 / 29
	max	1.2 min	3 h	3 h
time – tc constr.	Σ	6 min	-	6 min
	ϕ	0.6	-	0.5
	max	1.5	-	1.5
total computation time	Σ	2 h	5.2 h	7.2 h
	ϕ	11.7	2.9 min	36.4
	$Q_1/Q_2/Q_3$	1.5 / 3.5 / 14	71 / 72 / 80	1.6 / 5.1 / 31
	max	75.3	3 h	3 h

Table 10.6: Test case generation via ioco checking up to depth 20 for the PC_AS model. All values are given in seconds unless otherwise noted.

combined refinement and ioco check on these 79 model mutants. Actually, all of them are killed by our test suite, i.e., the fault coverage of our approximated test suite is not reduced for the given set of model mutants. Actually, it is even higher than that of the ioco test suite. This is due to the higher exploration depth we could reach with our combined approach (depth 30). The ioco check was limited to a depth of 20 and hence cannot kill 10 mutants that are revealed as non-conforming by our combined approach. Note that these 10 mutants are also not killed by other test cases generated by the ioco check.

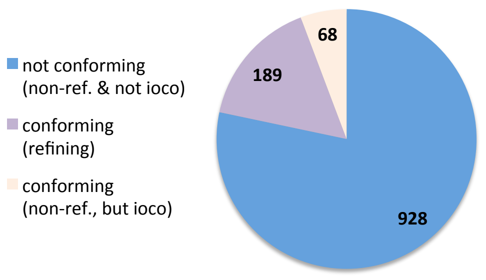
Our combined approach does not only result in a higher fault coverage on the model mutants. It is also faster than the ioco check as can be seen from Table 10.6. While our combined conformance check required approximately 5 hours, the ioco check took more than 7 hours. Note that we do not state runtimes without logs in Table 10.6 as it is only a small fraction of the total computation time for this example. The table also shows that the ioco check suffers from an outlier in the conforming mutants, which caused a runtime of 3 hours. This corresponds to our findings from the combined conformance check, where the ioco check also took very long for this particular mutant (cf. Table 10.5). For Ulysses, this mutant is even more problematic. It causes Ulysses to run out of memory on a machine with 8 GB RAM. For this reason, we did not finish the experiments with Ulysses and aborted after 4.7 hours and 318 finished mutants. Hence, less than 50% of the given model mutants were processed.

This model clearly demonstrates the benefits of our combined refinement and ioco check. These findings are also confirmed by the UML model of the particle counter as will be seen in the next section.

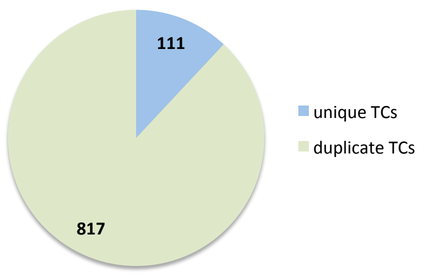
Complex Action System from UML Model

Due to the complexity of the PC_UML model generated by MoMuT::UML's frontend (cf. Table 9.1), we restricted the depth for the refinement check to 15 and the maximum depth for the subsequent ioco check to 5. Hence, the model is explored up to depth 20. Figure 10.17 shows the results of our combined refinement and ioco check. As can be seen from Figure 10.17a, 189 model mutants refine the original and 68 mutated models are conform as the found non-refinement does not propagate to an ioco violation. The remaining 78% of the mutants are non-conforming and result in 928 test cases. The vast majority of these test cases can be removed as they are duplicates of others. Hence, the test suite consists of 111 unique test cases. The lengths of these test cases are depicted in Figure 10.17c. The longest test case comprises 16 consecutive actions.

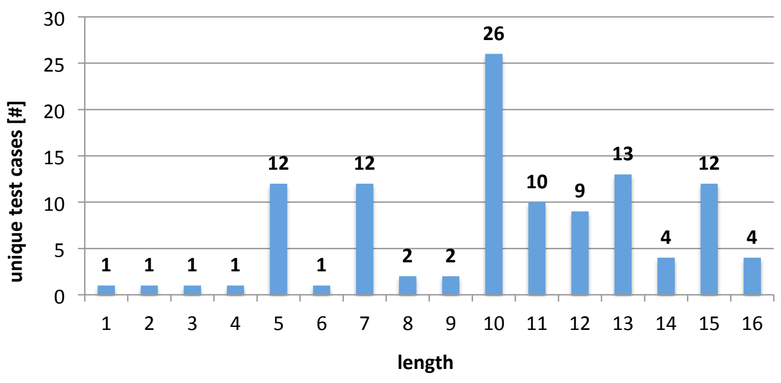
Figure 10.18 illustrates the number of steps required in the ioco checks to identify an ioco violation



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.17: Test case generation with our combined refinement and ioco check for PC_UML.

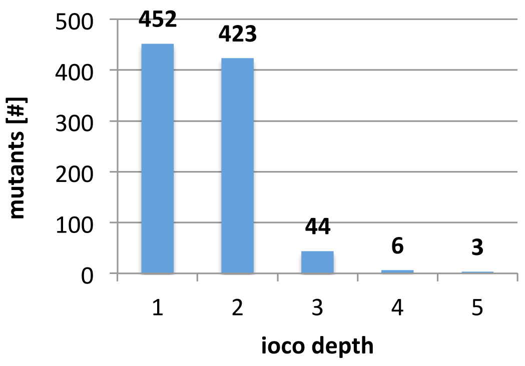


Figure 10.18: Overview of the ioco depths in our combined refinement/ioco check for PC_UML.

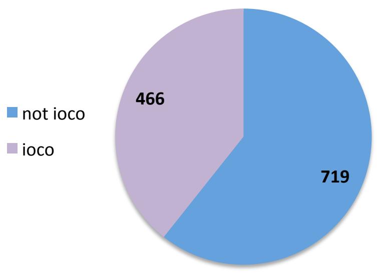
after the non-refinement. In most cases, a depth of 1 or 2 is sufficient. A depth of 3 is still required by 44 model mutants. Some were also found at depths 4 or 5.

Table 10.7 summarises the computation times for our combined conformance check. In this setting, the refinement check consumes the majority of the runtime. While 13.3 hours are spent on refinement checking, only 2.4 hours are used for the ioco check. However, for refinement we allowed for a search depth of 15, while the ioco check was restricted to a depth of 5. The overall computation time amounts to 16.2 hours.

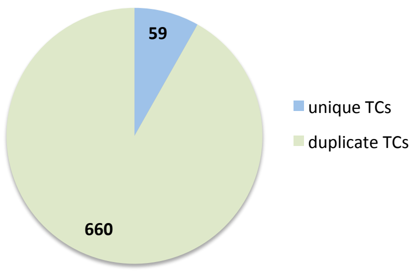
The stand-alone ioco check was performed with a maximum depth of 10. We experienced performance problems with ioco depths higher than 5 for the combined conformance check. However, we can

		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		189	68	928	1185
time – find states	Σ	-	-	-	13.5
time – ref. check	Σ	6.1 h	7.7	7.1 h	13.3 h
	ϕ	1.9	6.8 sec	27 sec	40 sec
	$Q_1/Q_2/Q_3$	0.2 / 0.23 / 3.8	2 sec / 6 sec / 6.5 sec	5 sec / 6 sec / 12 sec	5 sec / 8 sec / 17 sec
	max	4.3	1.8	3.9	4.3
time – ioco check	Σ	-	0.7 h	1.7 h	2.4 h
	ϕ	-	38 sec	7 sec	7.4 sec
	$Q_1/Q_2/Q_3$	-	6.5 sec / 31 sec / 51 sec	5.6 sec / 6 sec / 6.5 sec	5.5 sec / 6 sec / 6.5 sec
	max	-	2	27 sec	2
time – tc constr.	Σ	-	-	22.9	22.9
	ϕ	-	-	1.5 sec	1.2 sec
	max	-	-	3.7 sec	3.7 sec
total computation time	Σ	6.2 h	0.9 h	9.7 h	16.8 h
	ϕ	2	0.8	0.6	0.9
	$Q_1/Q_2/Q_3$	0.2 / 0.3 / 3.9	0.2 / 0.8 / 1.2	0.2 / 0.3 / 0.4	0.2 / 0.3 / 0.6
	max	4.4	2.2	4.1	4.4
total computation time without log	Σ	6.1 h	0.9 h	9.2 h	16.2 h
	ϕ	1.9	0.8	0.6	0.8
	$Q_1/Q_2/Q_3$	0.2 / 0.23 / 3.8	0.2 / 0.7 / 1.1	0.2 / 0.3 / 0.4	0.2 / 0.3 / 0.6
	max	4.3	2.2	4.1	4.3

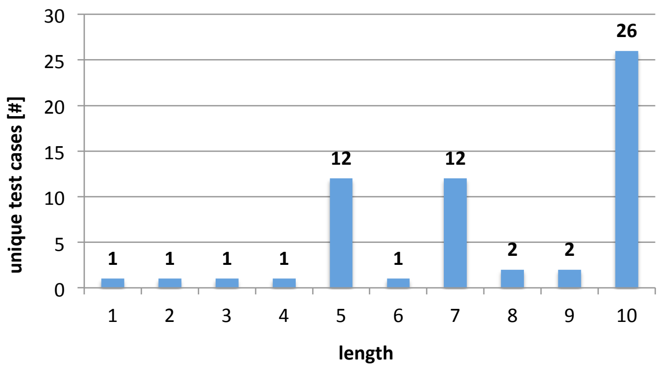
Table 10.7: Test case generation via the combined refinement/ioco check for the PC_UML model. The maximum search depth for the refinement check was set to 15, the subsequent ioco check was limited by a depth of 5, i.e., the overall maximal search limit was 20 steps. All values are given in minutes unless otherwise noted.



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Figure 10.19: Test case generation based on a stand-alone ioco check for PC_UML.

allow for this depth in the pure ioco check, which always starts from the initial state. The reason is that the particle counter model is very narrow in the beginning. The model starts with a sequence of three actions for reporting its state. Only then, the state space slowly widens. For our combined approach, the ioco check starts in deeper states and has to cope with a relatively larger branching factor, i.e., each state has many successor states.

Figure 10.19 summarises the results of the ioco check. For this model, the number of ioco-conform mutants is rather high (39%). This can partly be explained by the fact that we do not fully cover the model's state space. The remaining non-conforming mutants result in 719 test cases (cf. Figure 10.19a). Thereof, 92% are duplicates and only 59 unique test cases remain (Figure 10.19b). The majority of these test cases are 10 steps long. This is also the maximum length of the generated test cases as we restricted the ioco check by this limit. We analysed the sets of conforming and non-conforming mutants produced by the ioco check and the combined conformance check. The combined approach wrongly classifies 14 mutants as conforming. However, all of them are covered by other test cases that were generated by our combined conformance check. In turn, the ioco check missed 223 mutants due to the smaller search depth. Note that these 223 mutants cannot be killed by the ioco test suite. Hence, our combined approach achieves a higher mutation score on the model mutants than the ioco check.

Although the search depth was very limited compared to our combined approach, the ioco check takes considerably more time. While our combined approach took approximately 16 hours, the ioco check required 33.3 hours as can be seen from Table 10.8. Thereof, 22.8 hours were spent on checking 39% of the model mutants, which are the conforming mutants. On average, a non-conforming mutant requires 0.8 minutes, while a conforming mutant amounts to 2.9 minutes. As for PC_AS, the time required for writing logs is a negligible fraction compared to the overall runtime. Therefore, we did not

		not ioco	ioco	total
mutants [#]		719	466	1185
time – ioco check	Σ	9.8 h	22.8 h	32.6 h
	ϕ	0.8	2.9	1.7
	$Q_1/Q_2/Q_3$	4.6 sec / 0.3 / 1.6	2.8 / 2.9 / 3	0.3 / 2.4 / 2.8
	max	3.9	5.2	5.2
time – tc constr.	Σ	19	-	19
	ϕ	1.6 sec	-	1 sec
	max	5.8 sec	-	5.8 sec
total computation time	Σ	10.3 h	23 h	33.3 h
	ϕ	0.9	3	1.7
	$Q_1/Q_2/Q_3$	6.4 sec / 0.4 / 1.6	2.8 / 2.9 / 3	0.3 / 2.4 / 2.8
	max	3.9	5.2	5.2

Table 10.8: Test case generation via ioco checking up to depth 10 for the PC_UML model. All values are given in minutes unless otherwise noted.

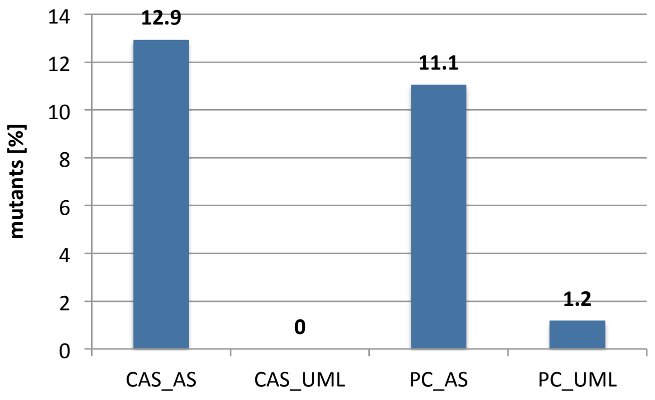


Figure 10.20: Percentage of mutants that are conforming according to our combined refinement and ioco check, but which are revealed to be not conforming in a pure ioco check.

state computation times without log file writing any more.

We also tried to generate test cases with Ulysses using the same search depth of 10. However, as for the PC_AS model, the memory consumption exceeded the available 8 GB of RAM and we aborted.

10.4 Discussion

We conclude this chapter by summarising the findings from our four experiments.

As pointed out in Section 10.2.1, our combined conformance check possibly misses some test cases as it might wrongly classify certain mutants as conforming, although they are not ioco-conform to the original model. Hence, it does not create a test case for these mutants and the generated test suite is an under-approximation of a full ioco test suite. As can be seen from Figure 10.20, this applied to almost 13% of the CAS_AS model mutants, while this was not the case for any of the CAS_UML model mutants. For PC_AS, 11% of the model mutants were missed, while for the PC_UML model the percentage was only 1.2%. Note that for the particle counter models, these numbers are lower bounds as the ioco checks were performed with lower exploration depths and hence classified some mutants as conforming, although they are possibly non-conforming in higher depths. However, it seems that

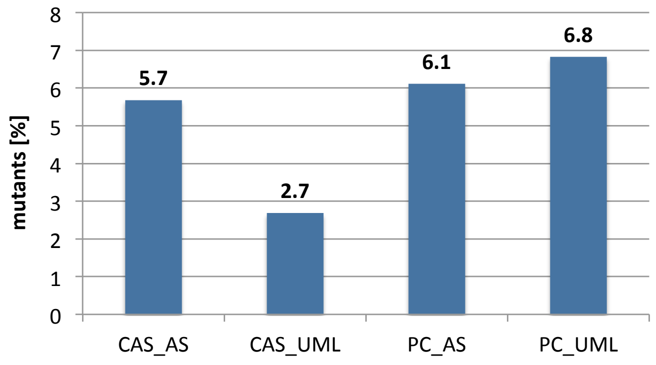


Figure 10.21: Percentage of non-refining model mutants that did not propagate to an ioco difference.

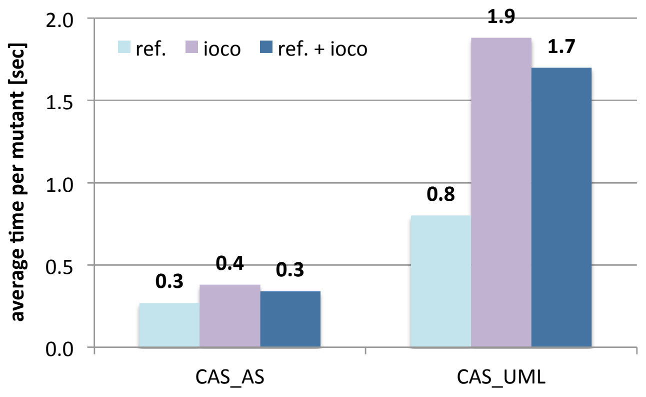


Figure 10.22: Comparison of the average computation times per mutant for the CAS. An exploration depth of 20 steps was used in each case.

the action systems generated from UML models have a beneficial structure that supports our approach. We investigated the CAS_UML model and found a plausible explanation: many intermediate results are saved in state variables and most mutations cause different states, which are detected by our refinement relation. Anyway, our under-approximation did in effect not lower the fault coverage of the generated test suites. For each model, all of the missed mutants were covered by other test cases generated by our combined conformance check.

Furthermore, the assumption that is underlying our idea of combining refinement and ioco holds for our use cases: non-refinement propagates to an ioco violation in most cases. The percentages of model mutants where this is not the case is depicted in Figure 10.21. For each model and the given set of mutants, less than 7% of the non-refining mutants do not violate the ioco relation after they have shown non-refinement. Hence, for more than 93% of the model mutants, our assumption holds. As can be seen from Figures 10.9, 10.12, 10.15, and 10.18, the propagation only requires up to 5 steps for the majority of the cases.

Finally, our experiments demonstrated that our combination of refinement and ioco is essential for complex models. For the simple CAS, our combined refinement and ioco check was only slightly faster than a pure ioco check. Figure 10.22 summarises the average time per model mutant for both CAS models. It compares the ioco check with our combined refinement/ioco check and also states the computation times achieved by our pure refinement check, which were presented in the previous chapter. For both CAS models, the ioco check is almost as fast as our combined conformance check. However, for the particle counter models, it turned out that our combined approach is essential in order to cope with such

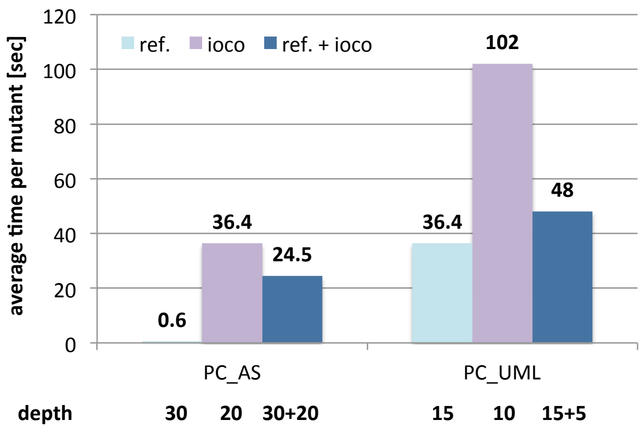


Figure 10.23: Comparison of the average computation times per mutant for the particle counter. Note that the exploration depths vary. They are stated at the bottom of the diagram.

complex models. Figure 10.23 compares the average computation time per model mutant for the particle counter models. Our combined approach reduces the computation time by more than 30% for the PC_AS model, and by more than 50% for the PC_UML model. Note that for the particle counter, the test suites generated by our combined approach turned out to be even stronger than those produced by the ioco checker. They achieve a higher mutation score on the given set of model mutants. The reason is that the exploration limits for the ioco checks were smaller. The used maximum depths are summarised at the bottom of the diagram.

Overall, we conclude that our combination is a valuable alternative to a stand-alone ioco check. We admit that for small systems like the CAS, it is not beneficial as a full ioco check is feasible. However, for complex systems like the particle counter, it can drastically decrease the computation time, while at the same time the exploration depths and hence the fault coverage of the mutated models can be increased. For the most complex model (PC_UML), the computation time was decreased by more than 50%, while 223 more mutated models were detected.

In the next chapter, we address the problem of the high redundancy in the generated test suites and conduct our final experiments, which include the execution of the generated test cases.

11 Final Optimisations and Experiments

The experiments with the particle counter presented in this chapter are part of the evaluation of the TRUFAL project and are joint work with AIT and AVL. The results will be documented in the TRUFAL Deliverable D5-1.

In this chapter, we describe our two last optimisation approaches and report on experimental results for the CAS and the particle counter including test case execution results.

11.1 Final Optimisations

11.1.1 Kill Check with Existing Test Cases

For MoMuT::UML's existing test case generation backend Ulysses, checking whether already generated test cases kill a given model mutant proved to be a valuable optimisation [10]. We adopted this approach for our backend. The main benefit is that we avoid the generation of redundant test cases. As pointed out in Chapter 8, our implementation generates a large set of test cases that contains many duplicates. For example, 80% of the test cases generated from the plain action system modelling the CAS were duplicates and could be removed after they had been generated. For the plain action system of the particle counter, the ratio was almost the same: 90% of the generated test cases were duplicates.

Instead of generating test cases and removing them later, we aim for the avoidance of redundancy in advance. The basic idea is to check whether an already existing test case kills a model mutant. If this is the case, no additional test case will be generated. If the model mutant cannot be killed, a new test case will be generated for this mutant. This corresponds to the test case construction strategy *S5* of the Ulysses backend [10].

In the following, we integrate this *kill check* into our combined refinement/ioco check, which was presented in the previous chapter. However, in general the concept is applicable to a pure refinement check or to a stand-alone ioco check as well. Algorithm 11.1 gives an overview of our kill check integrated into the refinement/ioco check. As input, it takes the original action system as and its initial state $init$, the set of mutated action systems $mutants$, the maximum depths for the refinement and ioco check ($maxRef$ and $maxIoco$ respectively), and finally the directory used for storing the test cases (tc_dir). At first, we calculate the reachable states for the refinement check (Line 1). This works analogously to

Algorithm 11.1 $tcgWithKillCheck(as, init, mutants, maxRef, maxIoco, tc\_dir)$

```
1: $states := findAllStates(as, init, maxRef)$
2: **for all** $asm \in mutants$ **do**
3:    **for all** $tc \in tc\_dir$ **do**
4:       $verdict := exec(tc, asm, init)$
5:       **if** $verdict = fail$ **then**
6:          **break**
7:       **end if**
8:    **end for**
9:    **if** $verdict \neq fail$ **then**
10:      $refIocoTcg(as, init, asm, states, maxIoco, tc\_dir)$
11:   **end if**
12: **end for**
```

Algorithm 10.1. For each mutant (Line 2), the test cases in the given directory are iterated (Line 3). The current test case tc is executed on the given mutant asm (Line 4). If it fails, we stop the kill check (Line 6). Hence, we do not necessarily execute all existing test cases, but stop when the first test case fails on the mutant. If no test case kills the mutant, i.e., in Line 9 the verdict is not *fail*, we start our test case generation (Line 10). The $reflocoTcg$ function performs our combined refinement and ioco check. In case of non-conformance, it constructs a test case. This test case is saved in the directory tc_dir, where it is available for the kill check performed on the following model mutants. In contrast to Algorithm 10.1, $reflocoTcg$ considers only one mutant and takes the pre-computed, reachable states as input. If a test case killed the mutant, i.e., the condition in the if statement in Line 9 evaluates to false, we do not generate an additional test case for this mutant.

The *exec* function used in Algorithm 11.1 relies on the ioco theory (cf. Section 3.2), where the execution of a test case TC on an implementation I is expressed by the synchronous parallel execution of the test case with the implementation – denoted as $TC \;\rceil\!| \; I$. An ioco test case is an LTS with inputs and outputs that has special properties (Definition 8.1). In the following, we denote this kind of LTSs as $TTS(L_I, L_O)$ (test transition systems). Furthermore, ioco presumes weakly input-enabled implementations (cf. Section 3.2.1), i.e., implementations belong to the class of Input Output Transition Systems (IOTSs) (Definition 3.19) and are denoted by $IOTS(L_I, L_O)$. Remember that L_I represents the input alphabet and L_O denotes the output alphabet. In this work, we take the view of the SUT, i.e., inputs are inputs to the SUT and controllable by the tester, and outputs are outputs from the SUT, which are observable by the tester. The synchronous parallel execution is a function $TTS(L_I, L_O) \times IOTS(L_I, L_O) \to LTS(L_I, L_O \cup \{\delta\})$ where the resulting LTS represents the executed sequence of events, i.e., the test log of the test run. $TC \;\rceil\!| \; I$ is defined by the following inference rules [192]:

$$\frac{i \xrightarrow{\tau} i'}{t \rceil\!| i \xrightarrow{\tau} t \rceil\!| i'}\ (1) \qquad \frac{t \xrightarrow{a} t' \quad i \xrightarrow{a} i'}{t \rceil\!| i \xrightarrow{a} t' \rceil\!| i'}\ a \in L_I \cup L_O\ (2) \qquad \frac{t \xrightarrow{\delta} t' \quad i \xrightarrow{\delta}}{t \rceil\!| i \xrightarrow{\delta} t' \rceil\!| i}\ (3)$$

Note that the states of TC and I are represented by t and i respectively. Unprimed states (t and i) denote the start state of a transition, primed states (t' and i') represent the end state of a transition. Initially, t and i are the initial states of the test case and the implementation respectively. Rule (1) deals with internal actions in the implementation. The according transition is consumed and the test case, which does not contain internal actions, does not move on. Rule (2) expresses synchronisation on common events. As the SUT is input-enabled, it accepts all inputs from the test case. Vice versa, the test case accepts all possible outputs from the SUT (cf. Definition 8.1). Hence, no deadlocks can occur. Finally, Rule (3) deals with quiescence (Definition 3.16). If the test case allows for quiescence, and the SUT does neither provide internal nor output actions, then the δ-action in the test case is consumed and the implementation remains in its current state. Test execution stops if a terminal state in the test case is reached. Remember that every terminal state in a test case represents a verdict (Definition 8.1). The verdict of the test run is determined by the reached terminal state's verdict.

Note that the implementation may behave non-deterministically. As a consequence, different test runs of the same test case on the same implementation may result in different verdicts. An implementation passes a test case if and only if all possible test runs do not lead to a fail verdict. Hence, when dealing with non-deterministic implementations, each test case must be executed several times in order to reveal all possible non-deterministic behaviours of the implementation. Note that this only works if the fairness assumption, which is an assumption of the ioco theory, holds (cf. Section 3.2.1). It states that a non-deterministic implementation eventually shows all of its non-deterministic behaviours.

In our case, we check whether a test case kills a given mutated action system. Hence, the implementation is represented by an action system, which we explore on the fly during test case execution to gain its LTS transitions. In principle, this exploration has already been described in Section 8.1. However, note that the model mutant is not necessarily input-enabled. Hence, we make the LTS input-enabled by angelic completion, which causes unknown inputs to be always accepted, but ignored (cf. Section 3.2).

Furthermore, we have to consider non-determinism in the model mutant. In contrast to testing a real implementation, we are not reliant on the re-execution of the test case under the fairness assumption as described above. As we explore the state space of the model mutant, we can systematically test all possible non-deterministic behaviours of the model mutant. In Prolog, this is implemented straightforwardly via Backtracking. If all of the behaviours in the model mutant lead to a pass or inconclusive verdict, the test case does not kill the model mutant. We move on to the next test case, or if all test cases have already been executed on the mutant, we start our mutation-based test case generation to create a new test case for this mutant (provided that the mutant is not equivalent). Note that the above rules assume explicit fail verdicts in the test cases. However, fail verdicts are implicit in our test cases. Hence, we reach a fail verdict whenever none of the above rules applies and we are not in a pass or inconclusive state. If this is the case for one of the non-deterministic behaviours of the model mutant, the mutant is killed by the test case and we do not generate an additional test case.

11.1.2 Combination with Random Test Cases

In addition to the rather costly mutation-based test case generation, we also implemented a random test case generation. Therefor, we explore the original action system and randomly choose one of the enabled actions. In this way, we generate a random trace through the model, which we extend to a test case like we construct test cases from traces to unsafe states (cf. Chapter 8). The generation of a random test suite requires two parameters from the user: the number of desired random test cases and their length, i.e., the number of actions leading to a pass verdict.

MoMuT::UML offers an option to combine random test case generation with model-based mutation testing. It first generates a set of random test cases as described above. Then, a kill check is performed as described in Section 11.1.1. Finally, the surviving model mutants are used as input for the more complex mutation-based test case generation. In principle, this concept works for arbitrary test suites. For example, a given set of manually designed test cases or a test suite based on the transition coverage of the original test model or the implementation can be used instead of a random test suite.

In the following sections, we report on experiences with purely mutation-based test suites with activated *kill checks*, with pure random test suites, and with the combination of random and model-based mutation testing as described above – again with activated *kill checks*. We execute these test suites on a set of faulty implementations to evaluate their effectiveness. We start with the CAS use case.

11.2 Experiments with the Car Alarm System

11.2.1 Test Case Generation

We generate test cases for the CAS in the final MoMuT::UML setting, i.e., we generate test cases from the complex action system generated by MoMuT::UML. This model was referred to as CAS_UML in the previous chapters. Furthermore, we reuse the 162 model mutants already created for our earlier experiments. We use our combined refinement and ioco check, which we presented in the previous chapter, with search depths of 20 steps each. However, we now additionally perform the *kill check* described in Section 11.1.1, i.e., we check whether already generated test cases kill a model mutant and only generate a new test if this is not the case. This leads to a mutation-based test suite, which we call M in the following.

Furthermore, we generate a random test suite as described in Section 11.1.2, which we refer to as R in the following. We use a similar setting as in former experiments with Ulysses [10], i.e., since the

	M	*R*	*C*
max. expl. depth for ref.+ioco	20 + 20	-	20 + 20
max. expl. depth for rand.	-	150	150
random tests [#]	-	3	1
model mutants [#]	162	162	162
survived model mutants [#]	17	31	17
test cases [#]	16	3	7
overall computation time [min]	7	0.5	8
time for finding states [sec]	5	-	5
avg. time – generate TC [sec]	4.8	10.7	5.3
avg. time – survived mutant [sec]	5.6	-	5.8
avg. time – kill mutant with TC [sec]	1.9	-	2.3

Table 11.1: Test case generation via the combined refinement/ioco check with activated kill check for the CAS_UML model.

CAS model is cyclic, we generate three long random test cases: one with 50 steps of random traversal, one with 100 steps, and one with 150 steps. However, note that the experiments with Ulysses were based on a different model of the CAS, which was deterministic and only allowed one particular sequence of the actions for activating the alarms (first the flash is turned on, then the sound). Similarly, the order for turning the alarms off again was fixed in the previous model: the sound is turned off first, followed by the deactivation of the flash lights. In contrast, our version of the CAS model is more abstract as it allows arbitrary orders of these actions – granting some implementation freedom.

Finally, we combine random testing with mutation testing. We will call the resulting test suite *C* in the following. Again, we use a similar setting as in the former Ulysses experiments [10]. We generate one test case based on a random traversal of 150 steps. Subsequently, we start our combined refinement and ioco check with search depths of 20 steps each. It also performs the *kill check*, i.e., it checks whether already existing test cases (including the random test) kill a model mutant.

Table 11.1 states the most important metrics on the three test suites. The first two rows summarise the exploration depths, which were already described above. Note that the depth required for finding all states of the CAS_UML model is 17 steps (cf. Table 9.1). Hence, our specified exploration depth of 20 for the refinement check allows for the full exploration of the state space of the model. The next two rows are a recap of the number of random tests and the number of model mutants. Note that for the random test suite *R*, these mutants were not used for the test case generation. However, we determined how many of these mutated models could not be killed by the random test suite. The number of survived model mutants is stated in the next row. Both *M* and *C* cannot kill 17 mutants. Clearly, test suite *R* has the highest number of survived model mutants. For *M* and *C*, the model mutants that did not survive were either killed by an existing test case or they led to a new test case. Test suite *M* consists of 16 test cases, *R* comprises 3 test cases, and *C* contains 7 test cases (one random and 6 mutation-based test cases). Note that test suite *C* is smaller than test suite *M*, but detects the same number or even more of the model mutants compared to the larger test suite *M*. Hence, the combination of random testing with mutation-based testing improves the quality of the generated test suites with respect to model mutation coverage and with respect to the size of the test suite.

The lengths of the test cases in test suite *M* are shown in Figure 11.1. The longest test case consists of 13 consecutive actions. There are no short test cases with length 1 or 2. This is due to the performed *kill check*. The mutations that lead to such short test cases were covered by longer test cases that had been generated before. For test suite *C*, the lengths of the test cases are depicted in Figure 11.2. The maximum length of the test cases is determined by the random test, which has a length 150 steps. Again, short test cases were not generated due to existing longer test cases. For test suite *R*, the lengths of the

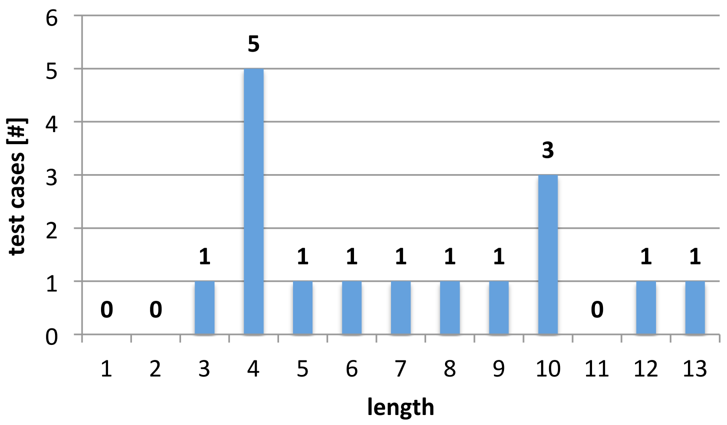


Figure 11.1: Overview of the lengths of the test cases in test suite *M* for the CAS.

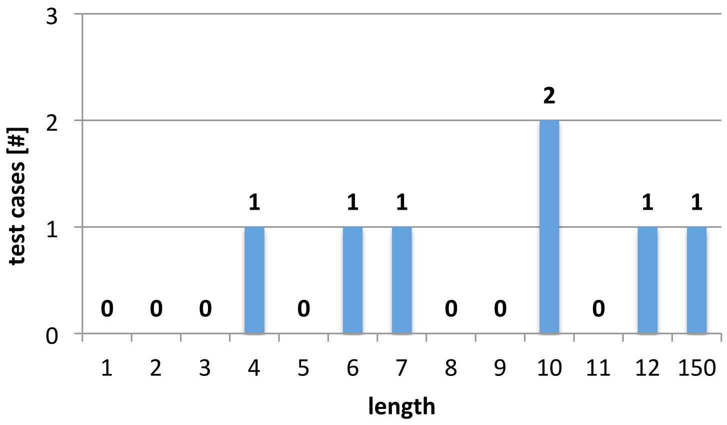


Figure 11.2: Overview of the lengths of the test cases in test suite *C* for the CAS.

test cases are 50, 100, and 150 steps as specified as input for the random test generation. An exemplary test case for the CAS has already been shown in Figure 8.2.

The subsequent rows of Table 11.1 report on the required computation times. The overall computation time states the overall time required for generating each test suite. The generation of the test suites involving conformance checking show almost equally long runtimes: approximately 7 minutes for test suite *M* and 8 minutes for test suite *C*. Unsurprisingly, the random test case generation is much faster than our mutation-based test case generation. The three random test cases could be generated in half a minute. One part of the overall computation time for test suite *M* and *C* consists of the time for the state space exploration, which forms the basis of our refinement check for all mutants (cf. Section 7.1.3). It is only a small fraction of the overall computation time (around 5 seconds). Finally, we state the average time required to generate one test case for each test suite. For *M* and *C*, we furthermore give values for the average time to process a surviving mutant. Finally, we calculated the average time required to kill a mutant with an existing test case. On average, the check whether an existing test case kills a model mutant requires less than half of the time needed to generate a test case. Unfortunately, the computation times of these experiments are not directly comparable to the computation times of our previous experiments presented in Section 10.3. The reason is that the computer used for the previous experiments was not available any more. Hence, the results presented above were generated on a MacBook Pro with an Intel i7 dual-core processor (2.8 GHz) and 8 GB RAM with a 64-bit operating system. This hardware is a little slower than the PC used for the previous experiments, which was equipped with an Intel i7 quad-core processor with 3.4 GHz.

In the following, we discuss the execution of our generated test suites on different faulty implementations of the CAS to assess their fault detection capabilities.

11.2.2 Test Case Execution

System under Test and Test Driver

For test case execution, we reuse a Java implementation of the CAS, which has been developed for the evaluation of test suites generated by MoMuT::UML's existing backend Ulysses [10]. Note that this implementation is deterministic, i.e., it fixes the order for turning the alarms on and off. The flash is always turned on before the sound, and the sound is always turned off before the flash.

The implementation defines Java interfaces for the SUT and the environment to facilitate the interaction between SUT and test driver. The interface of the SUT declares a method for each of the signals *Lock*, *Unlock*, *Close*, and *Open* specified in the testing interface (cf. Figure 1.3). Since the implementation uses simulated time, the interface also declares a *tick* method that is called by the environment, i.e., the test driver, to communicate to the SUT that one unit of time has passed. The test driver implements the environment interface, which declares callback methods for turning the flash and sound on and off as well as for arming and disarming the system (cf. Figure 1.3).

The test driver parses the abstract test cases, which have to be provided in the Aldebaran format. In case of a controllable action, it calls the corresponding method of the SUT. The SUT can respond to the test driver by means of callback methods. These calls are recorded by the test driver. Whenever the test driver parses an observable action, this action is compared to the oldest recorded call of a callback method. If the labels and optional parameters match, the execution of the test case is continued. Otherwise it is aborted and the execution of this test case results in a *fail* verdict. As already mentioned, the implementation uses simulated time. For actions incorporating the passage of time, the tick method is called repeatedly. If the specified amount of time, i.e., a number of ticks, has passed, the test driver continues. If no unspecified behaviour could be detected and the end of the test case is reached, the test run is finished. Remember that each sink state in a test case is a verdict state, i.e., *pass* or *inconclusive* (cf. Chapter 8).

Faulty SUTs

We used classical program mutation testing (cf. Section 4.2) to evaluate the effectiveness of the different test suites. Therefor, we reuse a set of 38 faulty SUTs from previous experiments [10], which were generated with μJava (version 3) [155]. All traditional mutation operators (method-level operators) have been applied. In total, μJava created 72 mutated implementations. After careful manual inspection, 8 of these mutated implementations were found to be equivalent to the original program. As the mutation score (Definition 4.9) is based on the number of non-equivalent mutants, the equivalent mutants were removed. Further 26 mutated implementations were found to be redundant, i.e., equivalent to other mutated implementations. In order to achieve a meaningful mutation score, they have been removed as well. In this way, 38 different faulty implementations of the CAS remained.

Results

Figure 11.3 summarises our results. The three bars on the left-hand side show the mutation scores (Definition 4.9) achieved by our three test suites when run on the 38 faulty implementations of the CAS. They are below 75% for each test suite. One reason for these rather low mutation scores is that many test runs result in inconclusive verdicts. For the survived faulty SUTs, 4 test cases (25%) of test suite *M* always resulted in inconclusive. For test suite *C*, also 4 test cases always led to inconclusive for the

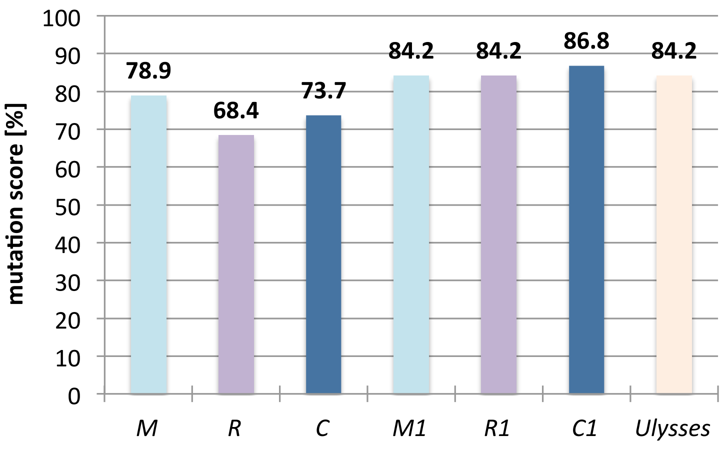


Figure 11.3: Mutation scores achieved by our test suites when executed on 38 faulty CAS implementations.

survived faulty SUTs. This is even a percentage of 57% of all tests in *C*. Furthermore, all 3 test cases of the random test suite result in inconclusive for the survived SUTs. This is due to the fact that we generate one linear test case (Definition 8.2) per unsafe state. Remember that the used model of the CAS allows for arbitrary orders for turning the alarms on and off. In contrast, the used implementations are deterministic, i.e., have fixed orders for these actions. If our linear test cases do not include the sequence as used in the implementation, but the other order, then our test cases are not executed to the end and result in inconclusive verdicts, which means that they do not kill.

For all of our test cases, we manually changed all sequences for turning the alarms on and off to the order used by the implementations. We named the resulting test suites *M1*, *R1*, and *C1* respectively. Thereby, no test run results in inconclusive any more and the mutation scores improved as can be seen from Figure 11.3. However, we still do not achieve a mutation score of 100%, i.e., we cannot detect all 38 non-equivalent mutated implementations. We cross-checked these results with a mutation-based test suite generated by Ulysses. As each model is fully explored, all non-equivalent model mutants are correctly identified. We used a setting for Ulysses, such that test cases are generated for each unsafe state of each non-equivalent mutant. Note that Ulysses creates adaptive test cases (Definition 8.3) and hence does not face the problem of inconclusive verdicts for this model. Even with this comprehensive test suite, not all faulty SUTs could be killed. We conclude that the set of model mutations used for our model-based mutation testing approach is not rich enough. The test suite *M1* and the test suite by Ulysses cannot detect the same 6 faulty SUTs. The best mutation score is achieved by *C1*, which incorporates one random test case that is able to kill one of the 6 faulty SUTs that were missed by *M1* and the Ulysses test suite. Hence, the combination of random and mutation testing is beneficial for this use case. This is in line with results from experiments with Ulysses [10] where the combination of random and mutation testing also showed to be advantageous. Test suite *R1* also misses only 6 faulty SUTs – however, not the same set as the other test suites.

From the above described results, we draw the following conclusions. First of all, for non-deterministic models, the generation of one linear test case per unsafe state potentially reduces the quality of the test suites. In particular, this is the case when the test cases are executed against a deterministic implementation. Note that when executed against a non-deterministic implementation, we could rerun our test cases until the SUT eventually shows the outputs specified in the test case that lead either to a pass or fail verdict – given that the fairness assumption holds (cf. Section 3.2.1). Furthermore, this experiment pointed out that the choice of the fault models, i.e., the model mutation operators is crucial, and that there is still room for improvement in the MoMuT::UML tool. Finally, our experiments with the CAS indicate that the combination of random and mutation-based test case generation is beneficial.

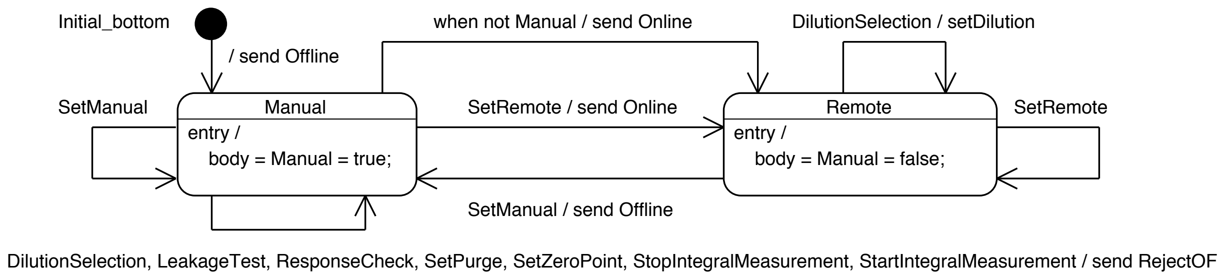

Initial_bottom
/ send Offline
when not Manual / send Online
DilutionSelection / setDilution
SetManual
Manual
entry /
body = Manual = true;
SetRemote / send Online
Remote
entry /
body = Manual = false;
SetRemote
SetManual / send Offline
DilutionSelection, LeakageTest, ResponseCheck, SetPurge, SetZeroPoint, StopIntegralMeasurement, StartIntegralMeasurement / send RejectOF

Figure 11.4: The orthogonal region modelling the communication mode changes of the particle counter.

The combined test suite achieved the highest mutation score on our 38 faulty SUTs. In the following, we report on results for the particle counter and check whether our conclusions drawn from the simple CAS use case also apply for a more complex system.

11.3 Experiments with the Particle Counter

11.3.1 Test Model

The experiments for the particle counter (cf. Section 1.6.2) with our final tool setting use an updated UML model. Modelling is usually an iterative process. Within the TRUFAL project, the UML model of the particle counter was created by AIT Vienna, while the use case itself comes from the industrial partner AVL. Hence, a lot of communication and clarification was required until a correct model was established. At the time of our previous experiments with the particle counter, the model was still subject to changes. In the meantime, it could be finalised. In principle, the model still represents the same functionalities and only small changes concerning details of the system behaviour were required. For example, the particle counter does not become busy at each change of the operating state in the updated model. Furthermore, the preliminary UML model was created with Papyrus MDT, which was the only UML editor supported by MoMuT::UML in the early phase of the TRUFAL project. Later on, support for Visual Paradigm (version 10.2) has been added and the final UML model of the particle counter has been created with Visual Paradigm. This was a requirement by AVL, who internally uses Visual Paradigm as their standard UML modelling tool.

As already explained in Chapter 9, a UML test model for MoMuT::UML comprises a class diagram specifying the testing interface (Definition 2.9), and a state machine modelling the system behaviour. In the final particle counter model, the state machine consists of three orthogonal regions: one for the operating state, one for switching between busy/ready, and one for the communication modes (manual/remote). The latter is shown in Figure 11.4. It is the simplest of the three regions. The orthogonal region modelling the operating state changes is the most complex region and exceeds an A4 page. A detailed description of the whole model goes beyond the scope of this thesis. However, to give an idea of the complexity of the model, we roughly describe the state machine in the following. The whole state machine consists of 19 states (5 nested into the top-level states and further 7 nested into these) and 39 transitions (excluding initial transitions). Transitions are triggered by signal receptions from the outside, by changes of internal variables, or by progress of time. As already mentioned in Chapter 9, the Object Constraint Language (OCL) is used to express guards on variables and on states of other regions. Furthermore, transition effects and entry/exit actions send outgoing signals or change the value of internal variables.

Similarly as for the intermediate particle counter model used in our previous experiments (cf. Table 9.1), we again estimate the approximate complexity of the model on the level of the generated action

	intermediate UML model	final UML model
actions [#]	109	139
state variables [#]	74	87
possible states [#]	$1.2 \cdot 10^{31}$	$2.7 \cdot 10^{36}$
mutated versions [#]	1185	3103

Table 11.2: Metrics describing the intermediate and the final UML models of the particle counter.

system. Table 11.2 relates metrics describing the intermediate UML model of the particle counter with the final UML model. According to these metrics, the model became more complex. The number of actions increased from 109 to 139. While the intermediate model defines 74 state variables, the final model uses 87 state variables. Considering the types of these state variables, which are either Boolean or bounded integers, the state space of the final model is larger than that of the intermediate model ($1.2 \cdot 10^{31}$ compared to $2.7 \cdot 10^{36}$ states). However, not all of these states are reachable from the initial state. For the intermediate model, we explored the system up to a depth of 25 and found out that there are 850 000 states actually reachable up to this depth and that there were still states to explore, i.e., this is a lower bound for the number of reachable states. This exploration required 4 days on a server machine with 30 GB RAM. This is also the reason why we did not repeat this analysis of the state space for the final model. The model metrics stated in Table 11.2 suggest that there are at least as many states reachable in the final model and that the exploration depth is again greater than 25 steps. For the intermediate UML model, MoMuT::UML generated 1185 mutated models. This value increased considerably for the final UML model. However, this is not solely caused by the model itself, but also by the fact that MoMuT::UML changed a mutation operator. Previously, effects of transitions were replaced by other possible effects leading to $O(n^2)$ mutations with n being the number of transition effects in the model. However, this mutation operator was only applied on transitions with effects. In the new version, this mutation operator also adds all possible effects to transitions that did not have an effect before, which considerably increased the number of mutants for the particle counter model.

Finally, note that the modelled behaviour is deterministic, i.e., there are no choices between output actions of the system. As a consequence, the generated test cases are all strictly linear without any inconclusive verdicts. Hence, the problem with linear test cases that include many inconclusive verdicts, which we encountered during execution on the deterministic CAS implementations (cf. Section 11.2.2), does not affect the particle counter case study.

In the following, we report on the test case generation for the final UML model of the particle counter. Therefor, we use the MoMuT::UML tool chain with the backend developed in the course of this thesis.

11.3.2 Test Case Generation

Like for the CAS, we again generated three test suites *M*, *R*, and *C*. The test suite *M* is mutation-based, the test suite *R* is randomly generated, and *C* is a combined random/mutation test suite. To generate *M* and *C*, we used our combination of refinement and ioco as presented in Section 10. Furthermore, the *kill check* presented in Section 11.1.1 has been activated.

The test case generation experiments were designed together with AIT, who also considerably contributed to the analysis of the results. Furthermore, the generation of all test suites was conducted at AIT, who dispose of a computer equipped with about 190 GB RAM and two 6-core Intel Xeon processors (3.47 GHz), running a 64-bit Linux (Debian 7.1). The system supports hyper-threading. Hence, 24 logical cores were available. To use the full capacity of such powerful systems, MoMuT::UML supports test case generation with multiple workers, which run in parallel. This can be easily achieved by partitioning the full set of model mutants into several sub-sets, which are processed in parallel. The user only needs to

	M	*R*	*C*
max. expl. depth for ref. + ioco check	18 + 7	-	20 + 5
max. expl. depth for rand.	-	25	20
random tests [#]	-	238	20
model mutants [#]	3103	3103	3103
model mutants killed by rand. TC [#]	-	2173	1819
survived model mutants [#]	552	930	683
test cases [#]	67 (+9)	238 (+2)	57 (+7)
overall computation time [h]	44.1	1.7	67.6
parallel backend workers [#]	21	1	21
time – gen. mutants [min]	14.2	-	15.5
time – gen. action systems [min]	13.1	0.02	12.0
time for finding states (ref. check) [h]	3.9	-	9.4
avg. time – generate TC [min]	12.7	0.43	26.8
avg. time – survived mutant [min]	22.7	-	24.4
avg. time – kill mutant with TC [min]	2.98	-	2.70

Table 11.3: Details on the test case generation for the mutation (M), random (R), and the combined (C) test suites for the particle counter.

state the desired number of workers. This parallelisation sometimes causes race conditions. Each worker uses the same directory for storing the generated test cases. If the kill check is activated, this directory contains the test cases that need to be checked for their ability to kill the currently processed mutant. If one worker saves a test case into the directory, while another mutant has already finished its kill check and also generates a test case, this can lead to duplicate test cases depending on the processed mutants.

Table 11.3 presents the most important metrics for each of the three test suites. The first two rows show the maximum exploration depths. As can be seen, the overall maximum depth for the mutation-based test case generation was 25 steps. However, the balance between the refinement and ioco depths varied. The rationale behind these varying depths was to examine the relative effect of these two bounds on the test case generation. In total, 3103 model mutants were generated. The test suites *R* and *C*, which both include random tests, are able to kill 70% and 59% of the model mutants with the random tests only. The different effectiveness of the random tests of these two test suites can be explained by the different lengths (25 vs. 20) and number (238 vs. 20) of the random tests. Considering all generated test cases, test suite *M* cannot kill 552 model mutants, test suite *R* cannot kill 930 mutated models, and 683 model mutants are not killed by test suite *C*. An examination of the test suites *M* and *C* showed that this is due to the reduced depth of the ioco check for *C*. This prevented *C* from finding 6 test cases, which could be generated by *M*. Note that not all of these survived mutants are necessarily equivalent mutants as we could not fully explore the state space of the model. The three test suites differ in the number of test cases and in the maximum length of the test cases. Test suite *M* comprises 67, *R* 238, and *C* 57 unique tests that can be executed on the SUT. In Table 11.3, the number in brackets states the number of duplicates in *M* and *C*, which are caused by the race condition between multiple worker threads generating tests in MoMuT::UML. For test suite *R*, the number in brackets states that 2 out of 240 generated random tests are not applicable to the SUT due to the abstraction of time in the model: the model assumes that some of the actions take no time, while in reality, the device might need some time to process these actions.

Figure 11.5 gives an overview of the lengths of the unique test cases in test suite *M*. The maximum length of the generated test cases is 19. There are no test cases with a length between 1 and 3 steps.

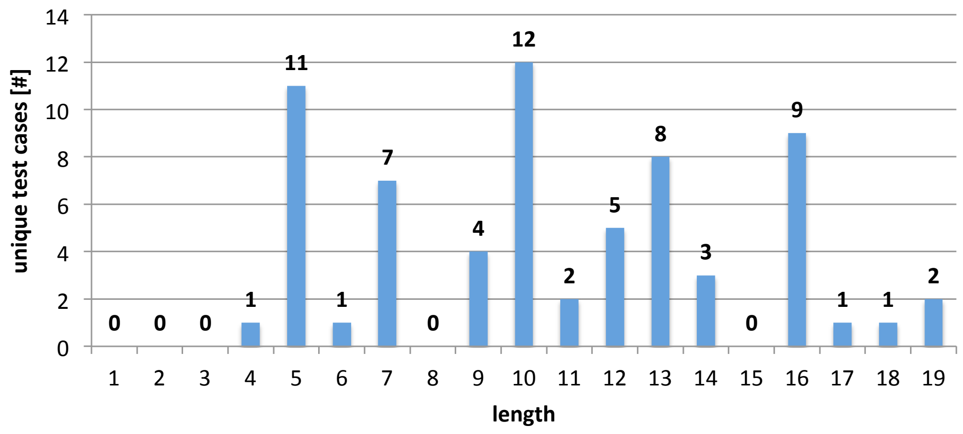


Figure 11.5: Overview of the lengths of the unique test cases in test suite *M* for the final particle counter model.

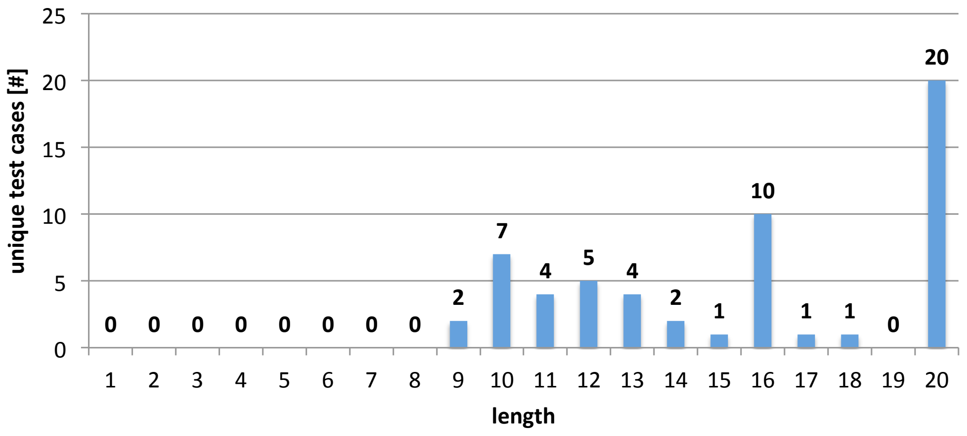


Figure 11.6: Overview of the lengths of the unique test cases in test suite *C* for the final particle counter model.

This can be explained by the *kill check*. All mutations in these depths are covered by longer test cases that have been generated before. For test suite *C*, the lengths of the unique test cases is depicted in Figure 11.6. Here, the maximum length is determined by the random test cases, which were limited to 20 steps. Again, there are no short test cases. In this test suite, the shortest test case has a length of 9 steps. The mutations in more shallow depths have all been covered by random test cases or by longer mutation-based test cases that have been generated before. For the random test suite *R*, all test cases have a length of 25 steps.

An exemplary test case for the final particle counter model is depicted in Figure 11.7. It is a direct graphical representation of the textual test case in Aldebaran format produced by MoMuT::UML. The test case does not contain inconclusive verdicts. As already pointed out, this is the case for all tests for the particle counter. Fail verdicts are implicit: every reaction from the SUT that is not specified in the test case leads to a fail verdict. Note that the first parameter of each action denotes time. For controllable actions, it states the number of time units the tester has to wait before sending the input to the SUT. For observable actions, it denotes the period of time in which the SUT may deliver a specified output.

Initially, the system is ready, in operating state *Pause – SPAU*, and offline. This is reflected by the first three events in the test case depicted in Figure 11.7. Then, it requests the system to switch to the *Standby* operating state. This entails a sequence of outputs from the system: it becomes busy, moves to

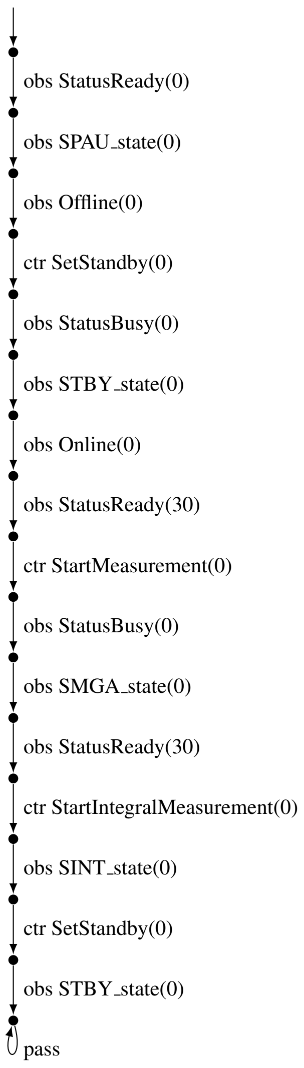


Figure 11.7: A sample test case for the particle counter.

operating state *Standby (STBY)*, switches to the remote mode (online), and finally becomes ready within 30 seconds. The next input to the SUT starts the measurement of the current particle concentration. Again, a sequence of observations similar to the previous one is triggered. Being ready and in operating state *measurement (SMGA)*, the system must accept the command that starts integral measurement, i.e., cumulative particle measurement. In this case, the system does not become busy, but directly switches to the according operating state (*SINT*). Finally, measurement is stopped by returning to the *Standby (STBY)* state.

The second part of Table 11.3 gives information on the required computation times. The overall computation times comprise the creation of model mutants in the form of OOASs, the mapping to action systems, and the actual test case generation performed by our backend. The latter takes advantage of 21 parallel workers for test suites *M* and *C*. Note that the computation times for *M* and *C* are not directly comparable due to the different exploration depths. The time required for mutant generation and the mapping of the OOASs to action systems are stated separately in the table. Note that for test suite *R*, no model mutants are generated and that hence only the original action system is translated into an action system. Furthermore, we state the time required for exploring the state space of the model up to the given depths for the refinement check. For test suite *M*, the model is explored up to depth 18. This requires

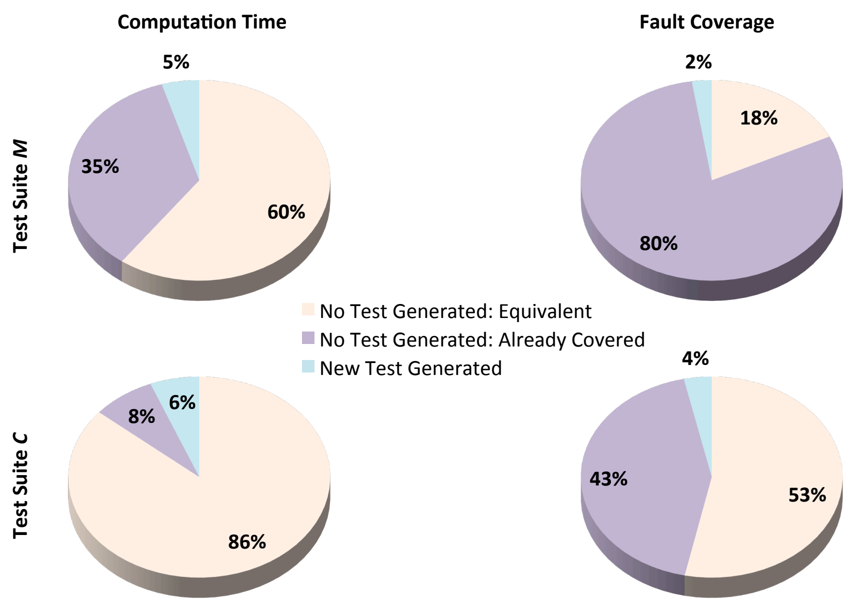


Figure 11.8: Test case generation for the particle counter: breakup of the computation time and the final fault coverage.

approximately 3.9 hours. For test suite *C*, the depth is 20 instead of 18, which increases the time to 9.4 hours. The last three rows give average values for the time required for creating one new test case, for processing a surviving mutant, and the time needed for checking whether an existing test already kills a model mutant.

Figure 11.8 shows a detailed breakup of the computation time and fault coverage for test suite *M* and the mutation-part of test suite *C*. More than 60% of the total test case generation time is spent on checking equivalent (surviving) model mutants. Note that this percentage is even higher for strategy *C* since the 20 random tests already kill many non-equivalent mutants, which are not considered in the mutation-based test case generation. Regarding fault coverage, it can be seen that for test suite *M*, 80% of the model mutants are covered by test cases generated from 2% of the model mutants. The combined test suite *C* shows a better ratio due to the random tests, which remove a lot of model mutants. However, test cases from 4% of the mutants still cover 43% of all model mutants. This data demonstrates that MoMuT::UML's mutation engine needs to be improved to generate more meaningful and less redundant model mutants.

Although we allowed for a refinement search depth of up to 20, none of the model mutants showed a refinement-depth greater than 17. Changing the maximum depth for the ioco check showed more effect. While almost all test cases were found at an ioco-depth of less than 3, there were 4 tests that needed a depth of 7 steps for the ioco check. Further analysis of the data indicates that our bounds were not high enough to find all possible test cases. For example, in some instances the random tests of *C* were able to kill mutants deemed equivalent by *M*.

We also attempted to generate test suite *C* using MoMuT::UML's existing backend Ulysses, but otherwise using the same setup. While our backend combining refinement and ioco proved to be CPU-bound, i.e., there was no danger of running out of RAM, Ulysses was unable to finish the task as the memory consumption of 21 parallel workers occasionally exceeded 190 GB. Even with many of the parallel computations being prematurely aborted, the computation time exceeded 144 hours.

11.3.3 Test Case Execution

System under Test

The three test suites are executed on a simulation of the particle counter device instead of a physical device, which is less complex and less expensive. Furthermore, it facilitates fault injection, which is used for the systematic evaluation of our test suites. AVL uses virtual testbeds, where the combustion engine as well as the measurement device are simulated by real-time software. The simulation of the particle counter is modelled in MATLAB Simulink[15] and compiled to a real-time executable. Thereby, two computers are used at AVL: one runs the simulation, the other one the test driver and the client for communication with the simulated SUT. In this way, the remote communication is included in testing. In this work, we test this simulation of the particle counting device. However, the generated test cases can of course also be executed on the physical device.

Concretion of the Abstract Test Cases and Test Execution Setup

As explained in Section 4.1.1, the abstract test cases derived from the test model need to be concretised in order to be executed on the SUT. For the CAS, a special test driver has been used that performs this adaptation on the fly during test execution. As the existing test infrastructure of AVL is based on concrete unit tests, a different approach has been taken for the particle counter: the abstract test cases are mapped to concrete test cases in the form of NUnit[16] test methods. Transforming the events of the abstract test cases into concrete method calls in the NUnit tests proved to be straightforward and has been implemented via simple XML-configured text substitutions. For example, the test case shown in Figure 11.7 is mapped to the C# test method shown in Listing 11.1. Controllable events are directly mapped to method calls of the SUT's interface. Observable events are not as simple to map. Since the SUT does not actively report state changes, observable actions have to be implemented via repeated polling of the system's state (these methods all start with *WaitFor*, e.g., *WaitForReady*). If the desired state is not reached within a specified time span, the method throws an exception causing a fail verdict. Unfortunately, repeated polling of the complete SUT state turned out to be too expensive. Therefore, our testing approach had to be *weakened* and only the state variables of the SUT related to the expected observable events are polled. As a consequence, the execution of a test will yield a pass verdict even if the SUT produces additional, unspecified outputs to the expected one. This change, which we could not circumvent within the project, limits the ability of our test suites to reveal failures via the coupling effect (cf. Definition 4.8). Furthermore, it implies that we cannot guarantee that the ioco relation holds between the SUT and the model – not even with exhaustive testing. Note that the translation of abstract into concrete test cases has been implemented by AVL and that the test cases were directly executed at AVL. The mapping of abstract to concrete test cases was implemented in the course of a bachelor thesis [26]. The thesis describes this mapping and the test execution setup in detail.

Test Execution Results on the SUT

The generated test cases were executed on the simulation of the device as described above and effectively revealed several failures. Although the simulation was in use for several years already, these tricky bugs had not been identified so far. In the following, we give an overview of the errors.

The first class of errors relates to changes of the operating state, which should not be possible when the device is busy. In the first case, the simulation allowed a switch from the operating state *Pause* to *Standby*, although the device was busy. The second issue concerned the activation of the integral

[15] `http://www.mathworks.co.uk/products/simulink` (last visit 2014-04-18)
[16] `http://www.nunit.org` (last visit 2014-04-18)

```
public void CMB_AVL489_MUTATION_guard_false__transition_9()
{
  avl489.WaitForReady(1);
  avl489.WaitForState(AVL489.States.Pause, 0);
  avl489.WaitForManual(1);
  avl489.Standby();
  avl489.WaitForBusy(1);
  avl489.WaitForState(AVL489.States.Standby, 0);
  avl489.WaitForRemote(1);
  avl489.WaitForReady(1);
  avl489.StartMeasurement();
  avl489.WaitForBusy(1);
  avl489.WaitForState(AVL489.States.Measurement, 0);
  avl489.WaitForReady(1);
  avl489.StartIntegralMeasurement();
  avl489.WaitForState(AVL489.States.IntegralMeasurement, 0);
  avl489.Standby();
  avl489.WaitForState(AVL489.States.Standby, 0);
}
```

Listing 11.1: The C# test case corresponding to the abstract test case of Figure 11.7.

	M	*R*	*C*
test cases [#]	67	238	57
execution time on original SUT [h]	0.5	1.6	0.5
execution time on all faulty SUTs [h]	7.9	27.4	8.1
survived faulty SUTs [#]	4	6	3
mutation score [%]	75	62.5	81.3

Table 11.4: Test case execution results for the faulty SUTs of the particle counter.

measurement of the number of particles over a period of time. If the device is measuring the current particle concentration and is still busy, the system must reject the request for cumulative measurement. However, the simulation accepted the command.

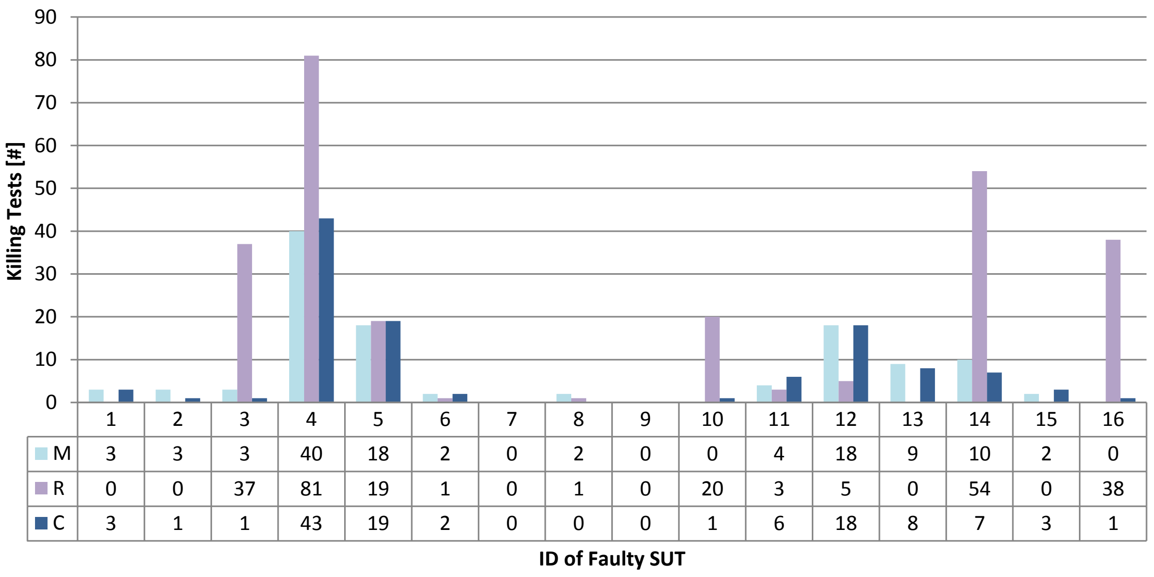
Further issues in the simulation were encountered by sending many inputs to the SUT in a short period of time. The simulation accepted them without any error messages, creating the impression that the inputs were processed. However, in reality, the inputs were absorbed and ignored. The correct behaviour would be to emit appropriate error messages.

However, not only the simulation of the device was erroneous but also the client for remote control of the device. The client did not correctly receive all error messages from the simulation. If the device is offline and receives the command to change the dilution value, the device returns the error message *RejectOffline*. In this particular case, the error messages were not recorded by the client.

Systematic Evaluation of the Test Suites

After the bugs described above had been fixed, a set of artificial faults were manually injected to evaluate the fault detection capabilities of the test suites *M*, *R*, and *C*. The artificial faults cover a range of easy-to-detect failures to very particular ones as will be seen later. In total, 16 faulty implementations have been prepared by AVL.

Two computers are needed for test case execution. The first one simulates the particle counter device



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
■ M	3	3	3	40	18	2	0	2	0	0	4	18	9	10	2	0
■ R	0	0	37	81	19	1	0	1	0	20	3	5	0	54	0	38
■ C	3	1	1	43	19	2	0	0	0	1	6	18	8	7	3	1

Figure 11.9: Evaluation results for the three test suites generated for the particle counter.

and runs the real-time simulation of the device, using about 300 MB RAM. The second one is running the test driver communicating with the simulated device. The test driver and the client for remote communication with the simulated measurement device require one core and about 1 GB of RAM. Both computers are running Windows 7 (64-bit). Table 11.4 summarises the test suite evaluation runs. As can be seen in the table, running the random test suite R took roughly three times as long as any of the two remaining test suites. Besides being most expensive when run, it also has the lowest mutation score (Definition 4.9). Note that we do not have equivalent SUT mutants as we did not apply mutation operators, but deliberately injected faults in the SUT. The combination of random and mutation tests achieved the best results in this evaluation. Test suite C does not only have the lowest number of test cases, but also the highest mutation score. The execution time on the faulty SUTs varies depending on how many tests fail. If a test case fails, the test run is finished after the fail verdict. If this happens at the beginning or in the middle of a test case, the test case is not fully executed to its end and needs less execution time.

Figure 11.9 highlights the results in more detail. In particular, the diagram shows that two of the faulty SUTs (number 7 and 9) were not found by any of the tests. These two artificial faults disable the ability to switch the system online in specific operating states, which can only be reached within a sequence of at least 8 and 9 steps. The faulty SUT with number 8 shows the same behaviour, but in an operating state that is easier to reach. Since the affected functionality is split across multiple regions, there is no simple model mutation directly emulating this type of problem. Hence, we identified a weakness in our model-mutation operators. Note that this was already identified in the evaluation of the test suites for the CAS. The missing model mutations cause our mutation-based test suites to miss test cases that cover these errors. At the same time, the failures were very unlikely to be revealed by random testing as they were buried deep in the model. A comparison of the test suites M and C revealed that C could have achieved an even higher mutation score if the depth for the ioco check had not been restricted to 5 (for the generation of test suite M a depth of 7 has been used). Furthermore, it was checked whether the test environment's inability to observe the full state of the SUT had adverse effects on test suites M and C, which was indeed the case. One example is faulty SUT number 16. It is not killed by test suite M due to the used test execution setup. Given full observability of the SUT's state, test suite M would kill this faulty SUT. Test suite C kills this mutant despite the restricted observations as one random test detects the failure.

11.4 Discussion

From our experiments with the particle counter, we conclude that fault-based test case generation is expensive, but it leads to high-quality test suites. Our test cases effectively revealed several subtle failures that have neither been found by manual testing nor by operation of the system over several years. Furthermore, mixing random and mutation-based test suites combines the best of two worlds: random tests are generated with relatively low efforts and mutation-based test case generation complements the test suite such that a given set of modelled faults is covered. That this combination is beneficial could already be seen from our experiments with the CAS (cf. Section 11.2.2). Furthermore, this confirms our earlier work with the Ulysses backend, where we came to the same conclusion [10].

Nevertheless, none of our three test suites was able to detect all injected faults – neither for the CAS nor for the particle counter. For the particle counter, the test execution environment is restricted due to technical reasons that we could not circumvent. We can only check for outputs expected by a test case, but not for unexpected outputs. This restriction decreases the fault detection capabilities of our tests.

Furthermore, we identified that none of our model mutants covers a particular faulty SUT of the particle counter. For the CAS, several faulty SUTs have not been detected due to missing model mutations. On the other hand, an analysis of the existing model mutants for the particle counter showed that too many similar model mutations are generated: 80% of the model mutants were killed by test cases generated from only 2% of the model mutants. Moreover, as in most mutation testing approaches, equivalent mutants are hindering: they consume 60% or more of the overall computation time. Hence, the choice of the fault models, i.e., the model mutation operators is crucial for the success of model-based mutation testing and there is still room for improvement in the MoMuT::UML tool. However, the mutation operators for UML models are implemented in the frontend and are not directly in the focus of this thesis.

Additionally to the applied model mutation operators, the quality of the modelled faults is also strongly influenced by the model itself. Depending on the used modelling style and model elements, different mutation operators are applicable leading to varying sets of model mutants. In turn, this results in different test suites with different fault detection capabilities [189]. Again, model creation is not the main topic of this work, but important for the success of our overall approach.

Finally, from the CAS case study, we found another issue that directly affects our test case generation tool. For non-deterministic models like the used CAS model, the generation of one linear test case per unsafe state potentially reduces the quality of the test suites. In particular, this is the case when the test cases are executed against a deterministic implementation. Note that when executed against an implementation that is as non-deterministic as the test model, we could rerun our test cases until the SUT eventually shows the outputs specified in the test case leading either to a pass or fail verdict – given that the fairness assumption holds (cf. Section 3.2.1). For the particle counter, this was not an issue as all test cases for this system are strictly linear and do not contain inconclusive verdicts. Nevertheless, this problem must be addressed in future work, e.g., by generating adaptive test cases similar as implemented by Ulysses or by a set of linear test cases, which is sufficient for deterministic SUTs.

12 Related Work

Parts of this chapter are based on our publications listed in Section 1.7.2.

Where appropriate, related work has already been discussed in earlier chapters. The mostly related work is the *Ulysses* tool, which was the original test case generation backend of the MoMuT::UML tool chain. It has already been described in Chapter 9. Furthermore, where appropriate, we referred to Ulysses throughout the thesis. Conformance relations related to our notion of refinement and to Input-Output Conformance (ioco) have been discussed in Chapter 3. Furthermore, formalisms related to action systems have been reviewed in Section 5.4. Regarding constraint-based test case generation, we already pointed out that many approaches rely on the SSA form (cf. Section 6.4.2). In the following, we give a broader overview of related work.

Automated test case generation is a wide and active field of research. A recent survey [24] gives an overview of the most prominent techniques for automated test case generation:

- structural testing using symbolic execution
- Model-Based Testing (MBT)
- combinatorial testing
- (adaptive) random testing
- search-based testing

Note that this list is not exhaustive and that techniques may be combined to form an automated software test case generation approach. As it is most relevant to this work, we review MBT, which is sometimes also called *specification-based testing*, in the next section. Subsequently, we concentrate on fault-based test case generation including both model-based approaches and white-box techniques.

12.1 Model-Based Testing

There exist various surveys [80, 84, 113, 5, 24] and books [56, 194, 209] on MBT. Furthermore, conferences and seminars are held on MBT, e.g., seminars at Schloss Dagstuhl [56, 52, 105].

There is no generally accepted classification of MBT approaches. The survey on automated test case generation [24] considers three main tracks followed in MBT:

- axiomatic approaches (including pre-/post-conditions)
- Finite State Machine (FSM) approaches
- Labelled Transition System (LTS) approaches relying on variants of ioco (cf. Section 3.2) and alternatively on alternating simulation [21] in the framework of interface automata

Hierons et al. [113] use a more fine-grained classification. They call the axiomatic approach described above *algebraic languages*, which are sometimes also referred to as abstract data types [84]. Furthermore, they consider *model-based languages*, which describe a system by means of possible states and operations that change these states, e.g., VDM [134], Z [184], or B [1]. These model-based languages possibly span an infinite state space – in contrast to *finite state-based languages*, which include FSMs and also Extended Finite State Machines (EFSMs) that incorporate additional internal data [148]. Moreover, LTSs are not solely considered as a category of their own, but as a formalism that can be used to describe the semantics for *process algebra state-based systems* like CSP [118, 175] or LOTOS [44]. Finally, *hybrid languages* for modelling systems that incorporate both discrete and continuous behaviour form a class of modelling languages. Aichernig et al. [84] introduce another category for Kripke structures and temporal logic for test case generation via model checking.

All of the above mentioned surveys do not only review different modelling techniques, but also give an overview of corresponding test case generation approaches. Furthermore, we refer to Fraser et al. [95] for more details on test case generation using model checkers, and to Lee and Yannakakis [148] for test case generation techniques from FSMs. In the following, we give a brief overview of related model-based testing tools.

12.1.1 Model-Based Testing Tools

At present, there exist numerous MBT tools – both non-commercial and commercial. An early overview of MBT tools was given by Hartman in 2002 [111]. Since then, more and more tools became available while others were stopped from being actively developed and maintained. In 2007, Utting and Legeard gave an updated overview [194]. In 2010, a review of model-based testing tool support has been published in the form of a technical report [182]. An online summary of open source/open binary tools for MBT was provided by Binder in 2012 [41].

In Section 4.1.3, we presented a taxonomy of MBT approaches by Utting et al. [195], which is based on several dimensions including the used modelling paradigm, test selection criteria, test generation technology, and test execution (cf. Figure 4.3). We already classified our approach using this taxonomy in Section 4.1.3. In the following, we briefly present related MBT tools and set them into context using the taxonomy. Thereby, we use the modelling formalism as main dimension.

Action Systems and Related Formalisms

Besides the Ulysses tool, we are not aware of any mutation-based test case generation approaches directly based on action systems. Some tools rely on similar formalisms. However, none of them uses mutations for test case generation.

In the course of our brief review on formalisms related to action systems (Section 5.4.4), we already mentioned *ProB* [152]. It is a simulator and model checker for the B-method including Event-B, which is closely related to the action system formalism. ProB can also be used for test case generation [203]. Two kinds of test purposes may be specified to guide test case generation: (1) either a predicate that has to be fulfilled in the end state of the test case (with limited length), or (2) a certain operation that has to be covered by the test cases. In this way, transition coverage can be specified as a test goal. Furthermore, the animation feature of ProB has been used indirectly for test case generation using test scenarios [156]. Both of the above mentioned test case generation approaches using ProB perform offline testing. Like the test case generator presented in this work, ProB is implemented in SICStus Prolog. For future work on our tool, it would be interesting to enhance our approach to support Event-B models.

Another formalism related to action systems is Circus (cf. Section 5.4.7). A test case generation tool for Circus specifications based on theorem proving has been presented recently [90]. Similarly to our approach, the formal semantics of the language is defined via UTP. An according testing theory is formalised for the Isabelle/HOL [164] theorem prover and integrated into an own testing environment for Circus. Two conformance relations are applied: traces inclusion and deadlocks reduction (cf. Chapter 3). Currently, the tool generates so-called symbolic tests for all traces up to a given length. Furthermore, each of these traces is instantiated once – either randomly or by SMT solvers like Z3 [76].

Furthermore, the SpecExplorer tool by Microsoft [198, 106] uses a modelling formalism that is comparable to action systems: model programs. They are state-oriented and guarded update rules are used to manipulate the state. Like the actions in our notion of action systems, these guarded update rules are labelled. In SpecExplorer, these labels are called actions and directly correspond to method invocations on a test adapter or on the SUT. Like our actions, SpecExplorer's actions may be parameterised and are distinguished between inputs and outputs. Another similarity to our work is that SpecExplorer also allows for non-deterministic models. However, in contrast to our approach, SpecExplorer is not mutation-based, but uses test purposes to restrict the state space to a finite subset that can be explored for test case generation. The underlying theory is based on interface automata and alternating simulation [21]. SpecExplorer supports both offline and online testing.

UML State Machines

In the following, we give a brief overview of MBT based on UML state machines to relate to the overall MoMuT::UML approach. For more details, we refer to a recent literature review [5].

A lot of research has been conducted on automated test case generation from UML state machines. However, none of these works is mutation-based. Indeed, there has been previous work on mutation testing of UML state machines, but in terms of model validation [89]. It seems that the MoMuT::UML approach is the first that actually generates test cases from mutated UML state machines.

One of the first tools for test case generation from UML state machines was based on various coverage criteria [167]. Following this lead, many other approaches also concentrated on coverage-based test case generation. For example, Kansomkeat and Rivepiboon flatten the UML state machines and generate test sequences covering all transitions [136]. In contrast to our MoMuT::UML, the used state machines are simpler as orthogonality is not supported. Fröhlich et al. [96] systematically transform use cases into UML state machines and generate test suites with a given coverage level by applying AI planning methods. Another tool that can be used for test case generation from UML state machines is AGATHA [39]. It accepts specifications written in different languages, e.g., UML. Each specification is transformed into an internally used Symbolic Transition System (STS) formalism. This translation is a similar concept as used in our MoMuT::UML tool chain, where UML models are transformed into action systems. STSs enrich LTSs with variables. In analogy to FSMs and EFSMs, AGATHA refers to STSs as EIOLTS (Extended Input Output Labelled Transition Systems). As a test generation technique, AGATHA uses symbolic execution to provide an exhaustive symbolic path coverage. To generate concrete input values for the symbolic paths, a constraint solver is used. Each symbolic path is considered as an equivalence class. Hence, they instantiate each symbolic path once, i.e., generate one set of satisfying assignments for each symbolic path.

Apart from coverage criteria, also test purposes have been employed for test generation from UML state machines. For example, Seifert [180] developed the TEAGER tool. The test purposes are based on given inputs and the tool calculates the according outputs to form test cases. The test purposes are either fixed input sequences or models using probabilities to represent varying behaviours of the environment. The tool offers offline test execution. Also Gnesi et al. derive test cases using test purposes [102]. The approach is founded on a formal semantics in the form of LTSs with inputs and outputs (cf. Definition3.12) and uses a conformance relation similar to ioco. In contrast to MoMuT::UML, OCL expressions, time triggers, and events with parameters are not supported.

Other approaches rely on random test generation. For example, Schwarzl et al. [177] generate random tests from UML state machines. They define extended symbolic transition systems (ESTSs), which extend STSs by incorporating timed behaviour via transition execution times and delay transitions. UML state machines can be transformed into ESTSs in a straightforward manner. In this work, only deterministic ESTSs are considered, the framework is based on alternating simulation. The tool uses a random strategy for test case generation and supports online and offline testing. Later on, they extended their approach to test purposes in the form of transitions that shall be covered by a test case [179]. In previous work, they used STSs and an according symbolic ioco testing theory [178].

Besides academia, also commercial companies provide tools for test case generation from UML state machines [194]. Smartesting CertifyIt [183, 149] combines constraint solving, proofs, and symbolic execution for test case generation. For test selection, the user may choose between various coverage criteria or may provide test purposes. The tool generates test cases in various formats that can be executed offline. In contrast to MoMuT::UML, it does not support non-determinism in the models. Conformiq Designer (formerly Qtronic) [72, 123] is based on an exploration algorithm to achieve a given coverage goal (requirements/structural coverage). Like Smartesting CertifyIt, the generated test cases are provided in numerous formats for offline execution and non-determinism is not supported. Automatic Test Generation (ATG) [58] is an add-on for IBM's UML tool Rational Rhapsody [125]. According to the product homepage [58], the given UML model is automatically analysed to gain information about the structure and behaviour of the model. *"This knowledge is used for the computation of a large number of test cases"* [58]. These test cases are executed offline.

Labelled Transition Systems

A tool directly based on the original ioco theory of Tretmans [191] is TorX [193]. Its successor tool JTorX [37] is implemented in Java and uses an updated ioco testing theory [192]. Both perform online testing and accept either directly LTSs in various formats or formalisms that can be mapped to LTSs, e.g., LOTOS process algebra specifications [44]. While TorX performs random testing, JTorx allows for user-defined test purposes. The algorithm implemented by TorX has been modified in order to work with symbolic models, so-called Symbolic Transition Systems (STSs), by Frantzen et al. [92].

A variant of the ioco conformance relation is used in TGV [126]. In contrast to TorX/JTorX, TGV is an offline testing tool. As specification language, TGV allows for various notions that have an underlying LTS semantics, e.g., LOTOS [44]. Test purposes are used to guide the test case generation. They are represented as LTSs with inputs and outputs with special states indicating whether to follow or truncate the exploration of the state space at this point. A follow-up tool of TGV is STG [68]. It works with a kind of STS, which is very similar to those introduced by Frantzen et al. [92] for TorX.

Furthermore, STSs were adopted by the AGATHA tool [39], which we already mentioned in the previous section on UML state machines. AGATHA is based on the symbolic execution of STSs. We followed a similar approach in our previous work [133]. However, instead of covering all symbolic paths, which is impractical for complex systems, we allowed the user to restrict the search space by test purposes. This was performed by calculating the synchronous product of the specification and a test purpose, which was then symbolically executed to generate test cases. Note that we used the STG tool mentioned above to produce the synchronous product.

Finally, we refer to the UPPAAL tools [112] that are focusing on model-based testing of real-time systems. They rely on timed automata and a variant of ioco, which is called relativised timed input output conformance (*rtioco*). The tool COVER performs offline testing based on test purposes or coverage criteria that are specified as observer automata. It is limited to deterministic systems. In contrast, TRON is an online testing tool especially targeting non-deterministic systems. Both tools rely on the UPPAAL model checking engine. In contrast to COVER and TRON, we only support a limited notion of time in our action systems.

12.2 Fault-Based Test Case Generation

In the previous section, we did not consider fault-based MBT approaches. We will discuss them here together with white-box mutation testing approaches.

12.2.1 Model-Based Mutation Testing

Model-based mutation testing has been applied to many modelling formalisms. One of the first models to be mutated were predicate-calculus specifications [59]. Test cases were applied on a set of mutated specifications to assess the fault detection capabilities of the test suite. In this way, the user was encouraged to generate test cases until all mutated specifications were found or until there were only equivalent specifications left. Hence, this approach is very similar to classical program mutation testing (cf. Section 4.2), but is based on specifications. Another early work in the field of model-based mutation testing was conducted by Stocks [187]. He defined a set of mutation operators for formal Z specifications, e.g., operator replacements, and presented criteria for the generation of distinguishing test cases. However, this approach has not been automated.

Later on, model checkers were available to check temporal formulae expressing equivalence between original and mutated models. In case of non-equivalence, this leads to counterexamples that serve as test cases [22, 42, 94]. This is very similar to our approach. However, in contrast to this equivalence check, we check for refinement and ioco respectively. Thus, we allow for non-deterministic models. Nevertheless, there also exist approaches using model checkers that take non-determinism and the involved difficulties into account. Okun et al. suggest to synchronise non-deterministic choices in the original and the mutated model via common variables to avoid false positive counterexamples [170]. Boroday et al. propose two approaches that cope with non-determinism: modular model checking of a composition of the mutant and the specification, and incremental test generation via observers and traditional model checking [46]. A further model-checker based test case generation approach that considers non-determinism [165] uses the model checker/refinement checker for Failures-Divergence Refinement (FDR) (cf. Section 3.1.3) for the CSP process algebra [175]. This work allows test case generation via test purposes, but not by model mutation. As already mentioned in Section 10.1, the first who used an ioco checker for model-based mutation testing were Weiglhofer and Wotawa. They worked with LOTOS specifications [200]. Later on, Ulysses has followed (cf. Section 9.1).

Our constraint-based refinement check is very similar to previous work on pre-/post-condition specifications. Aichernig and Pari Salas [19] set up a framework for model-based mutation testing using specifications in the form of contracts with pre- and post-conditions. They defined the underlying semantics via UTP designs. Conformance was defined in terms of refinement, which means in this setting that pre-conditions are weakened and post-conditions are strengthened (cf. Section 3.1.5). A tool has been implemented for OCL specifications based on a customised constraint solver. Later on, this approach has been adopted for Spec# specifications [139]. Spec# is a specification language by Microsoft that extends C#. It is also used in Microsoft's SpecExplorer tool mentioned above. Compared to our work, the above mentioned approaches dealt with transformational systems (cf. Definition 2.10). In contrast to reactive systems (cf. Definition 2.11) like action systems, they did not need to consider reachability issues.

The general theory behind model-based mutation testing with UTP's designs and refinement has been elaborated by Aichernig and He [12]. They present a so-called non-deterministic normal form that is very similar to our normal form for actions, which was introduced to allow for quantifier elimination (cf. Section 6.4.2). Furthermore, they instantiated the theory for a limited programming language having similarities with action systems. Instead of a do-od loop like in action systems, they allow for recursion. For the refinement check, recursion is considered up to a given upper bound.

Model-based mutation testing has also been applied to FSMs. For example, El Fakih et al. [87] represent many mutants in one *mutation machine* and generate test cases such that the considered mutations are covered. Hierons and Merayo [114] developed a model-based mutation testing theory for probabilistic FSMs that extend FSMs by assigning probabilities to the transitions. They present mutation operators and show how to generate distinguishing test cases. Furthermore, they use statistic sampling methods to estimate the probabilities of the SUT with sufficient precision. Later on, they extended their work to probabilistic and stochastic FSMs [115]. Using stochastic time, the time consumed between applying an input and receiving an output is given by random variables. Hence, it can be specified that an action will be performed within a certain amount of time units with a certain probability. The authors prove that deciding whether two probabilistic FSMs (or probabilistic stochastic FSMs respectively) are equivalent can be performed in polynomial time. Therefore, the equivalent mutants problem is not an issue in their approach.

Recently, colleagues adopted model-based mutation testing for real-time systems [18]. They rely on deterministic input/output timed automata, which are inspired by UPPAAL's timed automata [112]. As conformance relation, they use timed input-output conformance as introduced by Krichen and Tripakis [143]. The conformance check is implemented via bounded model checking with SMT solvers.

Another field of application for model-based mutation testing is security testing. For example, Wimmel and Jürjens [204] automatically generate test cases based on attack scenarios by applying mutation operators on formal security models for the AUTOFOCUS tool [122]. They use a constraint solver to find test sequences that satisfy mutated models represented as predicates over traces, but do not satisfy given security requirements. Dadeau et al. [74] use model-based mutation testing for security protocols. They present a set of mutation operators for the High-Level Protocol Specification Language (HLPSL) that are motivated by common protocol flaws. The mutation operators were applied on protocols supported by the AVISPA verification tool [25]. It has a rich library of security problems for certain protocols. For the given model mutants, it decides whether they are unsafe with respect to its known security problems. If this is the case, it delivers an attack trace exploiting the found vulnerability, which serves as a test case. The AVISPA tool uses an internal representation of the HLPSL specification that is basically an infinite-state transition system that facilitates formal analysis. The tool provides several backends employing different techniques, e.g., an on-the-fly model checker, a SAT-based model checker, and a constraint-logic-based attack searcher. The latter has been used by Dadeau et al. [74].

12.2.2 White-Box Approaches

Besides black-box testing, mutation-based test generation has also been conducted as white-box testing, i.e., on the source-code level, where no non-determinism has to be considered. In the early 1990s, Offutt [166] proposed a technique called *constraint-based test data generation*. It uses symbolic execution in order to generate constraints on the input values such that the path leading to the mutation (error) is found. Similarly, Wotawa et al. use the SSA form of Java-like programs to express their semantics and generate distinguishing test cases [208]. Brillout et al. generate C code from mutated Simulink models and distinguishing test cases are derived by the use of bounded model checking on the C-code level [51]. Furthermore, test case generation techniques based on constraints that do not consider mutations have been proposed. For example, Gotlieb et al. rely on structural criteria for test data generation via the SSA form. They work with constraint solving [103] and Constraint Logic Programming (CLP) [104].

As can be seen from our review of related work, there exist various works that are overlapping with our approach in one or several aspects. However, besides of Ulysses, we are not aware of automated test case generation approaches that are based on the action system formalism. Furthermore, no model-based mutation testing approaches were combining weak and strong mutation testing before as we proposed in Chapter 10.

13 Conclusion

In this thesis, we covered automated test case generation for reactive software systems. The chosen approach is called model-based mutation testing. In particular, we focused on the required conformance check. In this chapter, we provide a summary of the previous chapters including our main conclusions. Finally, we give an outlook on future work.

13.1 Summary and Conclusions

In the introduction, we gave an overview of the research context and requirements for this work that were partly predetermined by an existing tool chain called MoMuT::UML. Furthermore, we introduced two industrial use cases from the automotive domain, a car alarm system and a particle counting device, which were used throughout the thesis.

Next, we briefly discussed the software testing background and conformance relations. In particular, we focused on the conformance relations used in this work: refinement and Input-Output Conformance (ioco). We also gave an overview of related conformance relations.

The following chapter dealt with model-based mutation testing. First, we focused on model-based testing and discussed its advantages and possible drawbacks. Furthermore, we introduced classical program mutation testing. Finally, we presented the combination of these two concepts: mutation is used on the modelling level and test cases are generated such that all non-conforming model mutants are killed. Mutation testing relies on two assumptions: the competent programmer hypothesis and the coupling effect. Model-based mutation testing additionally assumes that the mutated models sufficiently represent realistic faults in the SUT. Hence, the success of model-based mutation testing highly depends on the model mutations. However, they are out of the scope of this thesis as they are provided externally by the frontend of the MoMuT::UML tool chain. Instead, this work focused on the second issue of model-based mutation testing: the conformance check that is required in order to generate distinguishing test cases.

Before the presentation of our conformance checking approaches, we introduced action systems as the modelling formalism used in this work. We gave an overview of classical action systems as known from the literature. Afterwards, we concentrated on the variant of action systems that is used in this work. We started with a subset of the language, which we called *plain action systems*, and extended this subset to the full language that needs to be supported in order to interact with the MoMuT::UML tool chain. The full action system language was called *complex action systems*. Typically, the semantics of action systems is expressed via weakest pre-conditions. In this work, we defined a predicative semantics that is close to a constraint satisfaction problem. This facilitates the use of modern constraint and SMT solvers. Finally, this chapter showed the relation of our predicative semantics to the weakest pre-condition semantics and gave an overview of formalisms that are related to action systems.

The following chapters concentrated on our refinement checking approach. Our predicative semantics for action systems was used to encode the refinement check between an original and a mutated action system as a constraint satisfaction problem. As action systems are reactive systems, we have to consider reachability issues. Therefore, we reduced the general refinement problem of action systems to a step-wise simulation problem only considering the execution of one iteration of the do-od block. To reduce the size of the constraints to be solved, we exploited the disjunctive structure of the do-od block of plain action systems. Although we achieved promising results for the smaller car alarm system case study, the more complex particle counter use case revealed performance problems.

Therefore, we developed a set of optimisations, which were presented and evaluated on the two use cases in Chapter 7. We experimented with different search strategies of the used constraint solver, applied syntactic instead of semantic mutation detection for the minimisation of the constraints, introduced the pre-computation of the state space in order to efficiently deal with a large set of mutated models, and finally applied incremental solving techniques. Overall, this lead to a very efficient implementation of our refinement checker for action systems. For our two use cases, we could reduce the computation times by more than 90% compared to our basic implementation.

If a mutated action system does not refine the original action system, our refinement checker provides a counterexample trace witnessing non-refinement. We extended our refinement checker to a test case generator, which creates test cases from the counterexamples. Therefore, we augment the given trace to the unsafe state with verdicts to form a valid test case. We generate so-called *linear test cases* that only consider one path to an unsafe state. If the system under test is non-deterministic, it may deviate from this path during execution. This is handled via *inconclusive* verdicts. They indicate that the system under test behaved correctly, but did not allow to reach the goal of the test case in this test run.

In the previous chapters, our refinement-based test case generator worked with plain action systems. For the integration into the MoMuT::UML tool chain, we needed to extend our tool to support the complex action systems produced by MoMuT::UML's frontend from the UML input models. By means of the car alarm system and the particle counter use cases, we demonstrated that MoMuT::UML's frontend introduces additional complexity into the action system models that need to be processed by our test case generator. In principle, this was not a surprise, but the actual dimension was astonishing. The complex action systems produced by MoMuT::UML are several orders of magnitude more complex than manually-designed plain action systems that model exactly the same behaviour. Hence, the transformations performed in MoMuT::UML's frontend should be reconsidered and investigated for potentials to reduce the complexity of the resulting action systems as this is an important factor for the overall performance of model-based mutation testing with MoMuT::UML.

The following chapter introduced a combination of refinement and ioco checking that was motivated by two reasons. First of all, despite its efficiency, our notion of refinement is not completely satisfying the needs of model-based mutation testing. The ioco relation is more suitable for various reasons, which we explained in detail. However, previous work has shown that ioco checking is rather demanding in terms of runtime and memory consumption. To counteract, we use our optimised refinement check as a preprocessing step for the efficient computation of an under-approximation of an ioco test suite. Instead of performing a full ioco check between the original and a mutated model, we first perform a refinement check. Only in case of non-refinement, the ioco check is initiated from the point where non-refinement has been detected. In this way, the subsequent ioco check is more targeted to those parts of the system that are actually affected by the mutation. Our combined conformance check is slightly weaker than a full ioco check. Hence, the generated test suite is an under-approximation of the test suite produced by a stand-alone ioco check. However, it turned out that our combined conformance check is a valuable alternative to a stand-alone ioco check. For small systems like the car alarm system, it is not necessarily beneficial as a full ioco check is feasible. However, for complex systems like the particle counter, it could drastically decrease the computation time, while at the same time the exploration depths and hence the fault coverage of the mutated models could be increased.

All of the experiments with our test case generator resulted in large test suites that contained a substantial amount of redundant test cases. We addressed this problem by performing a so-called *kill check*. It checks whether already existing test cases are able to kill a given mutant. If this is the case, no additional test case will be generated. Furthermore, we combined our mutation-based test case generation with relatively cheap random testing. The random tests served as initial test suite that already covers many mutated models. In this way, the more costly mutation-based test case generation needs to be performed only for the remaining mutants. This preserves fault detection on the mutated models, saves computation time, and also turned out to potentially increase the fault detection rate of the test suite if the model is too complex to be fully explored.

For our final experiments with the car alarm system and the particle counter, we generated three test suites each: a mutation-based test suite, a random test suite, and a random- and mutation-based test suite. We executed the resulting test suites on an implementation of the car alarm system and on the simulation of the particle counter that is used in daily business by our industrial partner. For the particle counter, several subtle failures that have neither been found by manual testing nor by operation of the system over several years have been revealed. Furthermore, we evaluated the fault detection capabilities of our test suites on several faulty systems. Although achieving acceptable mutation scores, none of the generated test suites was able to detect all introduced faults. This is due to several reasons that we reconsider in the following.

First of all, the model mutations implemented by MoMuT::UML's frontend are not comprehensive enough to model all possible faults in the implementations. On the other hand, our experiments showed that many model mutants lead to redundant test cases. Hence, the model mutations need to be revised. For the particle counter, the test execution environment had to be restricted due to technical reasons that we could not circumvent. In this way, the observations that could be made by the tester were restricted and decreased the fault detection capabilities of our tests. Finally, the car alarm system revealed an issue that directly affects our test case generation tool. For non-deterministic models like the car alarm system model, our strategy to generate one linear test case per unsafe state potentially reduces the quality of the test suites. In particular, this is the case when the test cases are executed against a deterministic implementation. Hence, future work will definitely include the generation of several linear test cases for one unsafe state or of adaptive test cases that incorporate all paths to an unsafe state in one test case.

Our test case generation tool is an improvement over MoMuT::UML's existing test case generation backend *Ulysses* in the sense that it can process models that are not feasible for Ulysses. For example, for our final experiments with the particle counter, we attempted to generate test cases with Ulysses using the same search depth. However, Ulysses was unable to finish the task as the memory consumption exceeded 190 GB. Hence, our tool makes a valuable contribution to automated test case generation for industrial-sized models. Our industrial partner AVL plans to use the tool in future projects.

Reconsidering the thesis statement made in the introduction, we conclude that the applicability of model-based mutation testing could indeed be improved by careful selection of the conformance relation, and by using symbolic techniques and modern constraint and SMT solvers to check it. In particular, we believe that the combination of our highly optimised refinement check with an ioco check is a promising approach. We believe that this thesis makes a valuable contribution to the field of model-based mutation testing. However, the problem is far from being solved completely, as discussed in the next section.

13.2 Future Work

Although we achieved satisfying results for our two industrial use cases, we are interested in further case studies to see how our tool works for different kinds of systems.

One of the most important items for future work is the improvement of our test case construction approach. As discussed above, our approach currently generates one linear test case per unsafe state. This causes problems during test case execution as pointed out in our final experiments with the car alarm system. As a first step, we plan to generate several linear test cases per unsafe state. This solves the problems for deterministic implementations. For non-deterministic implementations, it is more beneficial to produce adaptive test cases, which dynamically adapt to the actual outputs from the system under test. In this way, inconclusive verdicts are only issued if there is no possibility to reach the test goal any more.

A further branch for future work is to enhance our notion of refinement to better fit the needs of our combined refinement and ioco check. If we incorporate quiescence, distinguish between input and output actions, and consider input-enabledness of the system under test in our refinement check, we would strengthen our combined notion of conformance. If we furthermore consider all unsafe states that can be found by refinement, we would not under-approximate any more, but generate a test suite that is equivalent to a test suite generated by a stand-alone ioco check from the same set of mutants.

There are many other points in our approach that allow for experimentation. It would be interesting to compare our currently implemented breadth-first search with a depth-first search. This would lead to longer test cases. It should be investigated whether these longer test cases are more effective in detecting failures. However, if long test cases fail, the task of debugging is harder for the software engineers than if a short test case fails. This motivated our choice to start with a breadth-first search.

Furthermore, our conformance checking approach could delegate more work to the used constraint or SMT solver. For instance, we could adopt bounded model checking techniques and incrementally unroll the do-od block into one constraint system. In this way, we would not need to explicitly explore the states of the model. However, first experiments with the initially used constraint solver did not show promising results. In the meantime, we integrated Microsoft's SMT solver Z3, which may perform better for such large constraint systems. Furthermore, it would be interesting to investigate whether recent developments in model checking, such as IC3 [49], are applicable in our setting.

Moreover, there is also a lot of room for improvement in our implementation. We plan to integrate further constraint and SMT solvers and evaluate whether they are more efficient for our application. We also think that it would be beneficial for the overall MoMuT::UML approach to incorporate richer data types such as lists, which are already supported by Ulysses. Another interesting topic is the parallelisation of our implementation. MoMuT::UML offers a very basic parallelisation functionality by dividing the overall set of model mutants into subsets that are handed over to several instances of our test case generator. However, parallelisation could already be used earlier inside our test case generation tool. For example, the exploration of the reachable states or the checks whether the reached states are unsafe could be performed in parallel. Unfortunately, SICStus Prolog does not support parallelisation. Hence, it might be required to re-implement our tool in another programming language.

We furthermore plan to design a more general syntax for action systems that is not leaned against Prolog-specific notations. This would facilitate to directly use our action systems as a modelling language. Another possibility would be to extend our test case generator to be able to process Event-B models [2], which have many similarities with action systems.

Last but not least, we should turn to the frontend to further improve the overall model-based mutation testing approach implemented in MoMuT::UML. The model mutation operators need to be revised to reduce redundancies and to result in a representative set of faulty models. Furthermore, it should be investigated whether the transformations from the UML input models into action systems can be improved such that the action systems, which need to be processed by our test case generator, become less complex.

In conclusion, although the work presented in this thesis contributed to the state of the art in model-based mutation testing, there is still a lot of research to be done.

Bibliography

[1] Jean-Raymond Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. (Cited on pages 18, 56 and 161.)

[2] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. (Cited on pages 18, 56 and 170.)

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010. (Cited on page 56.)

[4] Allen Troy Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, 1980. (Cited on page 36.)

[5] Manuj Aggarwal and Sangeeta Sabharwal. Test case generation from UML state machine diagram: A survey. In *Proceedings of the 3rd International Conference on Computer and Communication Technology (ICCCT 2012)*, pages 133–140. IEEE, 2012. (Cited on pages 161 and 163.)

[6] Bernhard K. Aichernig. Model-based mutation testing of reactive systems - from semantics to automated test-case generation. In *Theories of Programming and Formal Methods- Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *LNCS*, pages 23–36. Springer, 2013. (Cited on page 40.)

[7] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In *Revised Selected Papers of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *LNCS*, pages 228–249. Springer, 2010. (Cited on pages 4, 8 and 102.)

[8] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient mutation killers in action. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation (ICST 2011)*, pages 120–129. IEEE, 2011. (Cited on pages 4, 8, 9 and 102.)

[9] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: a two-layered interpretation for testing. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011. (Cited on pages 4, 8 and 102.)

[10] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability (STVR)*, 2014. (Cited on pages 4, 9, 17, 43, 77, 97, 99, 102, 118, 119, 130, 143, 145, 146, 148, 149, 159 and 187.)

[11] Bernhard K. Aichernig, Harald Brandl, and Willibald Krenn. Qualitative action systems. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *LNCS*, pages 206–225. Springer, 2009. (Cited on pages 55 and 102.)

[12] Bernhard K. Aichernig and Jifeng He. Mutation testing in UTP. *Formal Aspects of Computing*, 21(1-2):33–64, 2009. (Cited on pages 59 and 165.)

[13] Bernhard K. Aichernig and Elisabeth Jöbstl. Efficient refinement checking for model-based mutation testing. In *Proceedings of the 12th International Conference on Quality Software (QSIC 2012)*, pages 21–30. IEEE, 2012. (Cited on pages 7, 8, 29, 41, 59 and 79.)

[14] Bernhard K. Aichernig and Elisabeth Jöbstl. Towards symbolic model-based mutation testing: Combining reachability and refinement checking. In *Proceedings of the 7th Workshop on Model-Based Testing (MBT 2012)*, volume 80 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 88–102, 2012. (Cited on pages 7, 29, 41, 59 and 75.)

[15] Bernhard K. Aichernig and Elisabeth Jöbstl. Towards symbolic model-based mutation testing: Pitfalls in expressing semantics as constraints. In *Workshops Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012)*, pages 752–757. IEEE, 2012. (Cited on pages 7, 29, 41 and 59.)

[16] Bernhard K. Aichernig, Elisabeth Jöbstl, and Matthias Kegele. Incremental refinement checking for test case generation. In *Proceedings of the 7th International Conference on Tests and Proofs (TAP 2013)*, volume 7942 of *LNCS*, pages 1–19. Springer, 2013. (Cited on pages 8, 29, 41 and 79.)

[17] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 2014. To appear. (Cited on pages 8, 79 and 93.)

[18] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In *Proceedings of the 7th International Conference on Tests and Proofs (TAP 2013)*, volume 7942 of *LNCS*, pages 20–38. Springer, 2013. (Cited on page 166.)

[19] Bernhard K. Aichernig and Percy Antonio Pari Salas. Test case generation by OCL mutation and constraint solving. In *Proceedings of the 5th International Conference on Quality Software (QSIC 2005)*, pages 64–71. IEEE, 2005. (Cited on page 165.)

[20] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 1–11. ACM, 1988. (Cited on pages 46 and 69.)

[21] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 1998)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998. (Cited on pages 28, 161 and 163.)

[22] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd International Conference on Formal Engineering Methods (ICFEM 1998)*, pages 46–54. IEEE, 1998. (Cited on page 165.)

[23] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. (Cited on pages 1, 11, 12, 14 and 37.)

[24] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. (Cited on page 161.)

[25] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of the 17th International Conference*

on Computer Aided Verification (CAV 2005), volume 3576 of *LNCS*, pages 281–285. Springer, 2005. (Cited on page 166.)

[26] Jakob Auer. Automated integration testing of measurement devices – a case study at AVL List GmbH. Bachelor's thesis, Graz University of Technology, 2013. `http://trufal.files.wordpress.com/2013/11/auerjakob_trufal_bachelorarbeit_final.pdf` (last visit 2014-04-18). (Cited on page 156.)

[27] Ralph Back, Martin Büchi, and Emil Sekerinski. Action-based concurrency and synchronization for objects. In *Proceedings of the 4th AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS 1997)*, volume 1231 of *LNCS*, pages 248–262. Springer, 1997. (Cited on page 56.)

[28] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1983)*, pages 131–142. ACM, 1983. (Cited on pages 2, 4 and 41.)

[29] Ralph-Johan Back and Reino Kurki-Suonio. Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988. (Cited on page 41.)

[30] 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 page 55.)

[31] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991. (Cited on pages 41, 42 and 60.)

[32] Ralph-Johan Back and Kaisa Sere. Action systems with synchronous communication. In *Proceedings of the Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 1994)*, volume A-56 of *IFIP Transactions*, pages 107–126. North-Holland, 1994. (Cited on page 41.)

[33] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR 1994)*, volume 836 of *LNCS*, pages 367–384. Springer, 1994. (Cited on pages 41, 42 and 43.)

[34] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. (Cited on page 18.)

[35] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard Version 2.0. Technical report, Department of Computer Science, The University of Iowa, September 2012. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf` (last visit 2014-04-18). (Cited on page 106.)

[36] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2nd edition, 1990. (Cited on pages 31 and 32.)

[37] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, volume 6015 of *LNCS*, pages 266–270. Springer, 2010. (Cited on pages 119 and 164.)

[38] Gilles Bernot. Testing against formal specifications: A theoretical view. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT 1991)*, volume 494 of *LNCS*, pages 99–119. Springer, 1991. (Cited on page 17.)

[39] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with AGATHA. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 591–596. Springer, 2003. (Cited on pages 163 and 164.)

[40] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999. (Cited on page 29.)

[41] Robert V. Binder. Open source tools for model-based testing, April 2012. `http://robertvbinder.com/open-source-tools-for-model-based-testing` (last visit 2014-04-18). (Cited on page 162.)

[42] Paul E. Black, Vadim Okun, and Yaacov Yesha. *Mutation Testing for the New Century*, chapter Mutation of Model Checker Specifications for Test Generation and Evaluation, pages 14–20. Kluwer Academic Publishers, 2001. (Cited on page 165.)

[43] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. Google Research Blog, June 2006. `http://googleresearch.blogspot.co.at/2006/06/extra-extra-read-all-about-it-nearly.html` (last visit 2014-04-18). (Cited on page 12.)

[44] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987. (Cited on pages 21, 162 and 164.)

[45] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction (MPC 1998)*, volume 1422 of *LNCS*, pages 68–95. Springer, 1998. (Cited on pages 4, 54 and 55.)

[46] Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3–19, 2007. (Cited on page 165.)

[47] Cari Borrás. Overexposure of radiation therapy patients in panama: Problem recognition and follow-up measures. *Revista Panamericana de Salud Pública*, 20(2-3):173–187, 2006. (Cited on page 1.)

[48] Jean-Louis Boulanger. *Static Analysis of Software: The Abstract Interpretation*. Wiley, 2011. (Cited on page 11.)

[49] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011. (Cited on page 170.)

[50] Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. Automated conformance verification of hybrid systems. In *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)*, pages 3–12. IEEE, 2010. (Cited on pages 4, 43, 95, 102 and 118.)

[51] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for Simulink models. In *Revised Selected Papers of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *LNCS*, pages 208–227. Springer, 2010. (Cited on page 166.)

[52] Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors. *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/portals/index.php?semnr=04371` (last visit 2014-04-18). (Cited on page 161.)

[53] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes (MOVEP 2000)*, volume 2067 of *LNCS*, pages 187–195. Springer, 2001. (Cited on page 26.)

[54] Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In *4th International Workshop on Formal Approaches to Software Testing (FATES 2004)*, volume 3395 of *LNCS*, pages 64–78. Springer, 2004. (Cited on page 26.)

[55] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975. (Cited on page 1.)

[56] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005. (Cited on pages 1, 33, 161, 181 and 183.)

[57] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, volume 6015 of *LNCS*, pages 150–153. Springer, 2010. (Cited on page 81.)

[58] BTC AG. IBM Rational Rhapsody Automatic Test Generation Add On. `http://www.btc-ag.com/de/SID-C29E7A70-8828B9BE/3006.htm` (last visit 2014-04-18). (Cited on page 164.)

[59] Timothy A. Budd and Ajet S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, 1985. (Cited on page 165.)

[60] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980. (Cited on page 36.)

[61] Michael J. Butler. *A CSP Approach to Action Systems*. PhD thesis, University of Oxford, 1992. (Cited on pages 46 and 49.)

[62] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP 1997)*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997. (Cited on pages 71, 74 and 79.)

[63] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement: An overview. In *PSSE 2004* [64], pages 1–17. (Cited on page 20.)

[64] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors. *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*. Springer, 2006. (Cited on pages 18, 19 and 175.)

[65] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988. (Cited on page 56.)

[66] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Pearson Education, 2007. (Cited on page 11.)

[67] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013. (Cited on page 81.)

[68] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 470–475. Springer, 2002. (Cited on pages 119 and 164.)

[69] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. (Cited on page 77.)

[70] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. (Cited on pages 46 and 69.)

[71] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999. (Cited on page 11.)

[72] Conformiq. Conformiq Designer. `http://www.conformiq.com/products/conformiq-designer` (last visit 2014-04-18). (Cited on page 164.)

[73] John Cooke. *Constructing Correct Software*. Springer, 2nd edition, 2005. (Cited on page 18.)

[74] Frédéric Dadeau, Pierre-Cyrille Héam, and Rafik Kheddam. Mutation-based test generation from security protocols in HLPSL. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST 2011)*, pages 240–248. IEEE, 2011. (Cited on page 166.)

[75] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge mathematical text books. Cambridge University Press, 2nd edition, 2002. (Cited on pages 17 and 18.)

[76] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. (Cited on pages 81 and 162.)

[77] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1998. (Cited on page 18.)

[78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978. (Cited on pages 34 and 36.)

[79] Richard A. DeMillo and A. Jefferson Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):109–127, 1993. (Cited on page 36.)

[80] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech 2007)*, pages 31–36. ACM, 2007. (Cited on pages 1, 33 and 161.)

[81] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. (Cited on pages 12 and 40.)

[82] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. (Cited on pages 19, 41 and 42.)

[83] Mark Dowson. The ARIANE 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997. (Cited on page 1.)

[84] Bernhard Aichernig (editor). MOGENTES Deliverable D1.2: State of the Art Survey – Part a: Model-based Test Case Generation Techniques. `https://www.mogentes.eu/public/MOGENTES_1-19a_1.1r_D1.2_Survey_Part-a.pdf` (last visit 2014-04-18), 2008. (Cited on pages 161 and 162.)

[85] Daniel Kroening (editor). MOGENTES Deliverable D3.2b: Modelling Languages (Final Version). `https://www.mogentes.eu/public/deliverables/MOGENTES_3-13_1.0r_D3.2b_ModellingLanguages.pdf` (last visit 2014-04-18), 2010. (Cited on page 102.)

[86] Georg Weissenbacher (editor). MOGENTES Deliverable D3.1b: Fault Models (Final Version). `https://www.mogentes.eu/public/deliverables/MOGENTES_3-09_1.0r_D3.1b_Fault_Models_Mutations.pdf` (last visit 2014-04-18), 2009. (Cited on page 102.)

[87] Khaled El-Fakih, Rita Dorofeeva, Nina Yevtushenko, and Gregor von Bochmann. FSM-based testing from user defined faults adapted to incremental and mutation testing. *Programming and Computer Software*, 38(4):201–209, 2012. (Cited on page 165.)

[88] Mats Carlsson et al. *SICStus Prolog User's Manual (Release 4.2.3)*. Swedish Institute of Computer Science, PO Box 1263, SE-164 29 Kista, Sweden, October 2012. `http://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf` (last visit 2014-04-18). (Cited on page 106.)

[89] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Paulo Cesar Masiero, Márcio Eduardo Delamaro, and W. Eric Wong. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE 1999)*, pages 210–219. IEEE, 1999. (Cited on page 163.)

[90] Abderrahmane Feliachi, Marie-Claude Gaudel, Makarius Wenzel, and Burkhart Wolff. The Circus testing theory revisited in Isabelle/HOL. In *Proceedings of 15th International Conference on Formal Engineering Methods (ICFEM 2013)*, volume 8144 of *LNCS*, pages 131–147. Springer, 2013. (Cited on page 162.)

[91] International Organization for Standardization. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework*. International Standard ISO/IEC 9646. International Organization for Standardization, Geneva, Switzerland, 1991. (Cited on page 17.)

[92] Lars Frantzen, Jan Tretmans, and Tim A.C. Willemse. Test generation based on symbolic specifications. In *Revised Selected Papers of the 4th International Workshop on Formal Approaches to Software Testing (FATES 2004)*, number 3395 in LNCS, pages 1–15. Springer, 2005. (Cited on page 164.)

[93] Lars Frantzen, Jan Tretmans, and Tim A.C. Willemse. A symbolic framework for model-based testing. In *Proceedings of the 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES 2006 and RV 2006)*, volume 4262 of *LNCS*, pages 40–54. Springer, 2006. (Cited on pages 26 and 29.)

[94] Gordon Fraser and Franz Wotawa. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, pages 16–22. IEEE, 2006. (Cited on page 165.)

[95] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability (STVR)*, 19(3):215–261, 2009. (Cited on page 162.)

[96] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *LNCS*, pages 472–492. Springer, 2000. (Cited on page 163.)

[97] Simson Garfinkel. History's worst software bugs. `http://archive.wired.com/software/coolapps/news/2005/11/69355` (last visit 2014-04-18), 2005. (Cited on page 1.)

[98] Christophe Gaston, Robert M. Hierons, and Pascale Le Gall. An implementation relation and test framework for timed distributed systems. In *Proceedings of the 25th International Conference on Testing Software and Systems (ICTSS 2013)*, volume 8254 of *LNCS*, pages 82–97. Springer, 2013. (Cited on page 26.)

[99] Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT 1995)*, volume 915 of *LNCS*, pages 82–96. Springer, 1995. (Cited on page 12.)

[100] Marie-Claude Gaudel. Checking models, proving programs, and testing systems. In *Proceedings of the 5th International Conference on Tests and Proofs (TAP 2011)*, volume 6706 of *LNCS*, pages 1–13. Springer, 2011. (Cited on page 12.)

[101] Ian P. Gent, Peter Nightingale, Andrew Rowley, and Kostas Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 172(6-7):738–771, 2008. (Cited on page 70.)

[102] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal test-case generation for UML statecharts. In *ICECCS*, pages 75–84. IEEE, 2004. (Cited on page 163.)

[103] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 53–62. ACM, 1998. (Cited on pages 46, 69 and 166.)

[104] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP framework for computing structural test data. In *Proceedings of the 1st International Conference on Computational Logic (CL 2000)*, volume 1861 of *LNCS*, pages 399–413. Springer, 2000. (Cited on page 166.)

[105] Wolfgang Grieskamp, Robert M. Hierons, and Alexander Pretschner, editors. *Model-Based Testing in Practice*, number 10421 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. `http://drops.dagstuhl.de/portals/index.php?semnr=10421` (last visit 2014-04-18). (Cited on page 161.)

[106] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Víctor A. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011. (Cited on page 162.)

[107] RAISE Method Group. *The RAISE development method.* BCS Practitioner Series. Prentice Hall, 1995. (Cited on pages 18 and 19.)

[108] Penny Grub and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing, 2nd edition, 2003. (Cited on page 1.)

[109] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977. (Cited on pages 34 and 36.)

[110] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer, 1985. (Cited on page 12.)

[111] Alan Hartman. Model based test generation tools. Technical report, AGEDIS Consortium, 2002. `http://www.agileconnection.com/sites/default/files/article/file/2012/XDD6047filelistfilename1_0.pdf` (last visit 2014-04-18). (Cited on page 162.)

[112] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *LNCS*, pages 77–117. Springer, 2008. (Cited on pages 26, 33, 119, 164 and 166.)

[113] 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):9:1–9:76, 2009. (Cited on pages 1, 18, 28, 32, 33 and 161.)

[114] Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic finite state machines. In *3rd Workshop on Mutation Analysis (Mutation 2007)*, pages 141–150. IEEE, 2007. (Cited on page 165.)

[115] Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software*, 82(11):1804–1818, 2009. (Cited on page 165.)

[116] Robert M. Hierons, Mercedes G. Merayo, and Manuel Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35–62, 2012. (Cited on page 26.)

[117] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on page 11.)

[118] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. (Cited on pages 49, 57 and 162.)

[119] C.A.R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall, 1998. (Cited on pages 19, 20, 26, 46, 51, 53 and 57.)

[120] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001. (Cited on pages 24 and 94.)

[121] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, 1982. (Cited on page 36.)

[122] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus: A tool for distributed systems specification. In *Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1996)*, volume 1135 of *LNCS*, pages 467–470. Springer, 1996. (Cited on page 166.)

[123] Antti Huima. Implementing Conformiq Qtronic. In *Proceedings of the 19th International Conference on Testing of Software and Communicating Systems (TestCom/FATES 2007)*, volume 4581 of *LNCS*, pages 1–12. Springer, 2007. (Cited on page 164.)

[124] Shamaila Hussain. Mutation clustering. Master's thesis, King's College London, 2008. (Cited on page 36.)

[125] IBM. Rational Rhapsody Family. `http://www-03.ibm.com/software/products/en/ratirhapfami` (last visit 2014-04-18). (Cited on page 164.)

[126] Claude Jard and Thierry Jéron. TGV: Theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005. (Cited on pages 1, 119 and 164.)

[127] Hannu-Matti Järvinen and Reino Kurki-Suonio. DisCo specification language: marriage of actions and objects. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS 1991)*, pages 142–151. IEEE, 1991. (Cited on page 55.)

[128] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. A novel method of mutation clustering based on domain analysis. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, pages 422–425. Knowledge Systems Institute, 2009. (Cited on page 36.)

[129] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 249–258. IEEE, 2008. (Cited on page 36.)

[130] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. (Cited on pages 34, 35, 36 and 39.)

[131] Elisabeth Jöbstl. Automating test case generation from transition systems via symbolic execution and SAT solving. Master's thesis, Graz University of Technology, 2009. (Cited on pages 8 and 29.)

[132] Elisabeth Jöbstl. Symbolic model-based mutation testing. `http://ejoebstl.files.wordpress.com/2011/07/fm_phd.pdf` (last visit 2014-04-18), 2011. Presented at the Doctoral Symposium of the 17th International Symposium on Formal Methods (FM 2011). (Cited on page 8.)

[133] 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 3rd International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 479–488. IEEE, 2010. (Cited on pages 8 and 164.)

[134] Cliff B. Jones. *Systematic software development using VDM*. Series in Computer Science. Prentice Hall, 2nd edition, 1990. (Cited on pages 18 and 161.)

[135] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing - a Contex-Driven Approach*. Wiley, 2002. (Cited on page 14.)

[136] 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 2003)*, pages 296–300, 2003. (Cited on page 163.)

[137] Joost-Pieter Katoen. Labelled transition systems. In Broy et al. [56], pages 615–616. (Cited on page 49.)

[138] James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233. ACM, 1975. (Cited on page 74.)

[139] Willibald Krenn and Bernhard K. Aichernig. Test case generation by contract mutation in Spec#. In *Proceedings of the 5th Workshop on Model-Based Testing (MBT 2009)*, volume 253(2) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 71–86. Elsevier, 2009. (Cited on page 165.)

[140] 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 *Revised Selected Papers of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009)*, volume 6286 of *LNCS*, pages 186–207. Springer, 2010. (Cited on pages 4, 43, 50, 55, 101 and 102.)

[141] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *Proceedings of the 11th International Workshop on Model Checking Software (SPIN 2004)*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004. (Cited on page 26.)

[142] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation. In *Proceedings of the 3rd International Colloquium on Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *LNCS*, pages 317–331. Springer, 2006. (Cited on page 26.)

[143] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009. (Cited on page 166.)

[144] Benjamin Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994. (Cited on page 55.)

[145] Matt Lake. Epic failures: 11 infamous software bugs. `http://www.computerworld.com/s/article/9183580/Epic_failures_11_infamous_software_bugs` (last visit 2014-04-18), 2010. (Cited on page 1.)

[146] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994. (Cited on page 57.)

[147] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM International Conference On Embedded Software (EMSOFT 2005)*, pages 299–306. ACM, 2005. (Cited on pages 26 and 33.)

[148] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996. (Cited on pages 161 and 162.)

[149] Bruno Legeard and Arnaud Bouzy. Smartesting CertifyIt: Model-based testing for enterprise IT. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 391–397. IEEE, 2013. (Cited on page 164.)

[150] Grégory Lestiennes. *Contributions au test de logiciel basé sur des spécification formelles*. PhD thesis, Université de Paris-Sud, 2005. (Cited on page 26.)

[151] Grégory Lestiennes and Marie-Claude Gaudel. Test de systèmes réactifs non réceptifs. *Journal Européen des Systèmes Automatisés, Modélisation des Systèmes Réactifs*, 39(1-3):255–270, 2005. (in French). (Cited on page 26.)

[152] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008. (Cited on pages 56 and 162.)

[153] Wen ling Huang and Jan Peleska. Exhaustive model-based equivalence class testing. In *Proceedings of the 25th International Conference on Testing Software and Systems (ICTSS 2013)*, volume 8254 of *LNCS*, pages 49–64. Springer, 2013. (Cited on page 1.)

[154] Hans J. Litteck and Peter J.L. Wallis. Refinement methods and refinement calculi. *Software Engineering Journal*, 7(3):219–229, May 1992. (Cited on page 18.)

[155] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability (STVR)*, 15(2):97–133, 2005. (Cited on page 148.)

[156] Qaisar A. Malik, Johan Lilius, and Linas Laibinis. Scenario-based test case generation using Event-B models. In *Proceedings of the 1st International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*, pages 31–37. IEEE, 2009. (Cited on pages 56 and 162.)

[157] Aditya P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the 15th International Conference on Computer Software and Applications (COMPSAC 1991)*, pages 604–605. IEEE, 1991. (Cited on page 36.)

[158] Stefan Mohacsi and Johannes Wallner. A hybrid approach for model-based random testing. In *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2010)*, pages 10–15. IEEE, 2010. (Cited on page 1.)

[159] Carroll Morgan. Of wp and CSP. In *Beauty Is Our Business - A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science, pages 319–326. Springer, 1990. (Cited on page 46.)

[160] Carroll C. Morgan. *Programming from Specifications*. Series in Computer Science. Prentice Hall, 1990. (Cited on page 18.)

[161] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–196, 1991. (Cited on page 56.)

[162] Glenford J. Myers. *The Art of Software Testing*. Wiley, 3rd edition, 2011. (Cited on pages 1, 12 and 14.)

[163] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. (Cited on page 11.)

[164] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. (Cited on page 162.)

[165] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from CSP models. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC 2008)*, volume 5160 of *LNCS*, pages 258–273. Springer, 2008. (Cited on page 165.)

[166] A. Jefferson Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, 1991. (Cited on page 166.)

[167] A. Jefferson Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on The Unified Modeling Language - Beyond the Standard (UML 1999)*, volume 1723 of *LNCS*, pages 416–429. Springer, 1999. (Cited on page 163.)

[168] A. Jefferson Offutt and Stephen D. Lee. How strong is weak mutation? In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV 1991)*, pages 200–213. ACM, 1991. (Cited on page 37.)

[169] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering (ICSE 1993)*, pages 100–107. IEEE, 1993. (Cited on page 36.)

[170] Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with model checker: Insuring fault visibility. In *Proceedings of the International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, pages 1351–1356, 2003. (Cited on page 165.)

[171] Alexandre Petrenko, Nina Yevtushenko, and Jiale Huo. Testing transition systems with input and output testers. In *Proceedings of the 15th International Conference Testing of Communicating Systems (TestCom 2003)*, volume 2644 of *LNCS*, pages 129–145. Springer, 2003. (Cited on page 93.)

[172] Alexander Pretschner and Jan Philipps. Methodological issues in model-based testing. In Broy et al. [56], pages 281–291. (Cited on pages 29, 30, 31 and 33.)

[173] Heinz Riener, Roderick Bloem, and Görschwin Fey. Test case generation from mutants using model checking techniques. In *Workshops Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST 2011)*, pages 388–397. IEEE, 2011. (Cited on pages 46, 67 and 69.)

[174] Mauno Rönkkö, Anders P. Ravn, and Kaisa Sere. Hybrid action systems. *Theoretical Computer Science*, 290(1):937–973, 2003. (Cited on page 55.)

[175] Andrew W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998. (Cited on pages 19, 49, 57, 162 and 165.)

[176] Rupert Schlick, Wolfgang Herzner, and Elisabeth Jöbstl. Fault-based generation of test cases from UML-models - approach and some experiences. In *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2011)*, volume 6894 of *LNCS*, pages 270–283. Springer, 2011. (Cited on page 8.)

[177] Christian Schwarzl, Bernhard K. Aichernig, and Franz Wotawa. Compositional random testing using extended symbolic transition systems. In *Proceedings of the 23rd International Conference on Testing Software and Systems (ICTSS 2011)*, volume 7019 of *LNCS*, pages 179–194. Springer, 2011. (Cited on page 163.)

[178] Christian Schwarzl and Bernhard Peischl. Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems. In *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)*, pages 122–131. IEEE, 2010. (Cited on page 164.)

[179] Christian Schwarzl and Franz Wotawa. Test case generation in practice for communicating embedded systems. *Elektrotechnik und Informationstechnik*, 128(6):240–244, 2011. (Cited on page 163.)

[180] Dirk Seifert. Conformance testing based on UML state machines. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM 2008)*, volume 5256 of *LNCS*, pages 45–65. Springer, 2008. (Cited on page 163.)

[181] Emil Sekerinski and Kaisa Sere. A theory of prioritizing composition. *The Computer Journal*, 39(8):701–712, 1996. (Cited on pages 49, 55 and 107.)

[182] Muhammad Shafiq and Yvan Labiche. A systematic review of model-based testing tool support. Technical Report SCE-10-04, Carleton University, May 2010. `http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-10-04.pdf` (last visit 2014-04-18). (Cited on page 162.)

[183] Smartesting. Smartesting CertifyIt. `http://www.smartesting.com/en/product/certify-it` (last visit 2014-04-18). (Cited on page 164.)

[184] J.M. Spivey. *The Z Notation – a Reference Manual*. Prentice Hall, 2nd edition, 1992. (Cited on page 161.)

[185] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. (in German). (Cited on page 30.)

[186] Leon Sterling and Ehud Shapiro. *The Art of Prolog – Advanced Programming Techniques*. MIT Press, 2nd edition, 1994. (Cited on pages 71 and 82.)

[187] Philip Alan Stocks. *Applying formal methods to software testing*. PhD thesis, Department of computer science, University of Queensland, 1993. (Cited on page 165.)

[188] Stefan Tiran. The Argos manual. Technical Report IST-MBT-2012-01, Institute for Software Technology – Graz University of Technology, 2012. `https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=275803&pCurrPk=67399` (last visit 2014-04-18). (Cited on page 55.)

[189] Stefan Tiran. On the effects of UML modeling styles in model-based mutation testing. Master's thesis, Graz University of Technology, 2013. (Cited on page 159.)

[190] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992. (Cited on page 17.)

[191] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996. (Cited on pages 4, 21, 24, 25, 28, 93, 94 and 164.)

[192] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008. (Cited on pages 21, 27, 28, 32, 93, 144 and 164.)

[193] Jan Tretmans and Ed Brinksma. TorX: Automated model-based testing. In *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 31–43, 2003. (Cited on pages 1 and 164.)

[194] Mark Utting and Bruno Legeard. *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann Publishers, 2007. (Cited on pages 1, 14, 29, 30, 31, 32, 33, 161, 162 and 164.)

[195] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability (STVR)*, 22(5):297–312, 2012. (Cited on pages 1, 33, 34 and 162.)

[196] Michiel van Osch. Hybrid input-output conformance and test generation. In *Formal Approaches to Software Testing and Runtime Verification (FATES/RV)*, volume 4262 of *LNCS*, pages 70–84. Springer, 2006. (Cited on page 26.)

[197] Margus Veanes and Nikolaj Bjørner. Alternating simulation and IOCO. In *Proceedings of the 22nd International Conference on Testing Software and Systems (ICTSS 2010)*, volume 6435 of *LNCS*, pages 47–62. Springer, 2010. (Cited on page 28.)

[198] 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 *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008. (Cited on pages 1 and 162.)

[199] Martin Weiglhofer and Bernhard K. Aichernig. Unifying input output conformance. In *Proceedings of the 2nd International Symposium on Unifying Theories of Programming (UTP 2008)*, volume 5713 of *LNCS*, pages 181–201. Springer, 2010. (Cited on pages 25 and 26.)

[200] Martin Weiglhofer and Franz Wotawa. ”On the fly” input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 286–291. ACTA Press, 2008. (Cited on pages 118 and 165.)

[201] Stephan Weißleder. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. PhD thesis, Humboldt Universität zu Berlin, 2009. (Cited on page 1.)

[202] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the 38th annual Design Automation Conference (DAC 2001)*, pages 542–545. ACM, 2001. (Cited on page 81.)

[203] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proceedings of the 21st IFIP International Conference on Testing of Communicating Systems and the 9th International Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2009)*, volume 5826 of *LNCS*, pages 179–194. Springer, 2009. (Cited on pages 56 and 162.)

[204] Guido Wimmel and Jan Jürjens. Specification-based test generation for security-critical systems using mutations. In *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM 2002)*, volume 2495 of *LNCS*, pages 471–482. Springer, 2002. (Cited on page 166.)

[205] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971. (Cited on page 56.)

[206] Niklaus Wirth. *Programming in Modula-2*. Springer, 1982. (Cited on page 56.)

[207] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996. (Cited on page 57.)

[208] Franz Wotawa, Mihai Nica, and Bernhard K. Aichernig. Generating distinguishing tests using the Minion constraint solver. In *Workshops Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 325–330. IEEE, 2010. (Cited on pages 46, 67, 69 and 166.)

[209] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors. *Model-Based Testing for Embedded Systems*. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, 2011. (Cited on page 161.)

Appendix

A Full Action System Model of the Car Alarm System

The following listing shows the full Prolog source code of an action system modelling the CAS. Parts of it have been presented earlier in this work in Listing 5.1. A few minor differences to this earlier presentation are commented in the listing below. Firstly, the standard notation for non-deterministic choice (operator [ ]) and sequential composition (operator ;) have been used so far. In the concrete Prolog syntax below, Prolog's operators for conjunction and disjunction have been used as they naturally represent these concepts. Hence, Prolog's disjunction (; operator) represents non-deterministic choice. Prolog's conjunction operator (, operator) represents sequential composition. This notation stems from the original language designed in MOGENTES for the Ulysses tool [10]. A further minor difference to the presentation above concerns syntactic parts that are ignored by the tools developed throughout this thesis. They only concern Ulysses, which also works with this syntax.

```
% namespace "as" indicates that we deal with a specification model
% (namespace "asm" would be used for a mutated model)
:- module(as, [var/2, input/1]).
% library for constraint logic programming over finite domains
% (required for conditions in guards)
:- use_module(library(clpfd)).
:- public(as/0).
:- dynamic(as/0).
:- dynamic(type/2).

% type definitions
% ignore labeling part for integer-based types (relict from Ulysses)
type(enum_State, X) :- X in 0..7, labeling([],[X]).
type(int_0_4, X) :- X in 0..4, labeling([],[X]).
type(int, X) :- X in 0..270, labeling([], [X]).
type(bool, X) :- member(X, [true, false]).

% types of state variables
var([aState], enum_State).
var([fromAlarm, fromArmed], int_0_4).
var([fromSilentAndOpen, flashOn, soundOn], bool).

% state definition
state_def([aState, fromAlarm, fromArmed, fromSilentAndOpen, flashOn, soundOn]).

% initial state
init([6, 0, 0, false, false, false]).

% controllable actions
input(['Close', 'Lock', 'Open', 'Unlock']).

% observable actions
output(['ArmedOff', 'ArmedOn', 'FlashOff', 'FlashOn', 'SoundOff', 'SoundOn']).

% action system
as :-
% no methods used in this action system
  methods (
    none
  ),

% non-deterministic choice: denoted by [] previously in this work
%         is represented by Prolog's OR operator (semicolon ;)
% sequential composition: denoted by ; previously in this work
%         is represented by Prolog's AND operator (comma ,)
  actions (
    'ArmedOff'(Wait_time)::(fromArmed #= 1) =>
    (
      ((aState #= 4) => (fromArmed := 0))
     ;
      ((aState #= 1) => (fromArmed := 2))
    ),

    'ArmedOn'(Wait_time)::(true) =>
    (
       ((Wait_time #= 20 #/\ aState #= 3) => (aState := 2))
       ;
       ((Wait_time #= 0 #/\ aState #= 2 #/\ fromSilentAndOpen #= true) =>
            (fromSilentAndOpen := false))
    ),

    'Close'(Wait_time)::(true) =>
    (
      ((aState #= 6 #/\ fromAlarm #= 0) => (aState := 4))
     ;
      ((aState #= 5) => (aState := 3))
     ;
      ((aState #= 7 #/\ fromAlarm #= 0) =>
           (fromSilentAndOpen := true, aState := 2))
    ),

    'FlashOff'(Wait_time)::(flashOn #= true) =>
    (
      ((fromAlarm #= 1 #/\ Wait_time #= 0) =>
           (fromAlarm := 3, fromArmed := 0, flashOn := false))
     ;
      ((aState #= 0 #/\ fromAlarm #= 2 #/\ Wait_time #= 270) =>
           (aState := 7, fromAlarm := 3, fromArmed := 0, flashOn := false))
     ;
      ((fromAlarm #= 3 #/\ Wait_time #= 0) =>
           (fromAlarm := 0, fromArmed := 0, flashOn := false))
    ),

    'FlashOn'(Wait_time)::(flashOn #= false) =>
    (
      ((aState #= 1 #/\ fromArmed #= 2) => (fromArmed := 3, flashOn := true))
     ;
      ((aState #= 1 #/\ fromArmed #= 3) => (fromArmed := 4, flashOn := true))
    ),

    'Lock'(Wait_time)::(true) =>
    (
      ((aState #= 6 #/\ fromAlarm #= 0) => (aState := 5))
     ;
      ((aState #= 4 #/\ fromArmed #\= 1) => (aState := 3, fromArmed := 0))
    ),

    'Open'(Wait_time)::(true) =>
    (
       ((aState #= 4 #/\ fromArmed #\= 1) => (aState := 6, fromArmed := 0))
      ;
       ((aState #= 3 #/\ fromArmed #\= 1) => (aState := 5, fromArmed := 0))
    ;
      ((aState #= 2 #/\ fromSilentAndOpen #= false) =>
          (aState := 1, fromArmed := 1))
  ),

  'SoundOff'(Wait_time)::(soundOn #= true) =>
  (
      ((fromAlarm #= 1 #/\ Wait_time #= 0) =>
          (fromAlarm := 3, fromArmed := 0, soundOn := false))
    ;
      ((aState #= 0 #/\ fromAlarm #= 2 #/\ Wait_time #= 270) =>
          (aState := 7, fromAlarm := 3, fromArmed := 0, soundOn := false))
    ;
      ((fromAlarm #= 3 #/\ Wait_time #= 0) =>
          (fromAlarm := 0, fromArmed := 0, soundOn := false))
    ;
      ((Wait_time #= 30 #/\ aState #= 1 #/\ fromArmed #= 4) =>
          (aState := 0, fromAlarm := 2, fromArmed := 0))
  ),

  'SoundOn'(Wait_time)::(soundOn #= false) =>
  (
      ((aState #= 1 #/\ fromArmed #= 2) => (fromArmed := 3, soundOn := true))
    ;
      ((aState #= 1 #/\ fromArmed #= 3) => (fromArmed := 4, soundOn := true))
  ),

  'Unlock'(Wait_time)::(true) =>
  (
      ((aState #= 5 #/\ fromAlarm #= 0) => (aState := 6))
    ;
      ((aState #= 3) => (aState := 4))
    ;
      ((aState #= 7 #/\ fromAlarm #= 0) =>
          (aState := 6, fromAlarm := 0, fromArmed := 0))
    ;
      ((aState #= 2 #/\ fromSilentAndOpen #= false) =>
          (aState := 4, fromArmed := 1, fromSilentAndOpen := false))
    ;
      ((aState #= 1 #/\ fromArmed #= 4) =>
          (aState := 6, fromAlarm := 1, fromArmed := 0))
    ;
      ((aState #= 0 #/\ fromAlarm #\= 4) =>
          (aState := 6, fromAlarm := 1, fromArmed := 0))
  )
),

dood (
  'Close'(0)
; 'Open'(0)
; 'Lock'(0)
; 'Unlock'(0)
; [T1:int]:'ArmedOn'(T1)
; 'ArmedOff'(0)
; 'FlashOn'(0)
; [T2:int]:'FlashOff'(T2)
; 'SoundOn'(0)
; [T3:int]:'SoundOff'(T3)
),

qdes (none). % ignore (relict from Ulysses)
```

B Extended Tables

This appendix contains extended versions of some tables that stated arithmetic mean values. They are extended by values for the quartiles $Q_1/Q_2/Q_3$. Q_1 represents the first quartile, i.e., quantile with $q = 0.25$. Q_2 is the second quartile, i.e., the quantile with $q = 0.5$, which is the median. Q_3 represents the third quartile, i.e., quantile with $q = 0.75$. The most interesting values were already mentioned in the text describing the original tables.

		1: find mutated action	2: reach & non-refine	total
CAS_1	Σ	23	18	**41**
	ϕ	0.11	0.09	0.2
	$Q_1/Q_2/Q_3$	0.02/0.03/0.05	0.02/0.08/0.14	0.05/0.13/0.18
	max	13	0.37	13
CAS_10	Σ	160	19	**179**
	ϕ	0.77	0.09	0.86
	$Q_1/Q_2/Q_3$	0.02/0.03/0.06	0.02/0.09/0.15	0.05/0.13/0.19
	max	127	0.38	127
CAS_100	Σ	32.4 min	23	**33 min**
	ϕ	9.39	0.11	9.5
	$Q_1/Q_2/Q_3$	0.03/0.04/0.06	0.02/0.10/0.17	0.06/0.15/0.23
	max	28 min	0.49	28 min
CAS_1000	Σ	4.2 h	18	**4.2 h**
	ϕ	73	0.09	73
	$Q_1/Q_2/Q_3$	0.02/0.04/0.05	0.02/0.08/0.14	0.05/0.12/0.17
	max	3.4 h	0.35	3.4 h

Table B.1: Extended version of Table 6.1 (values for quartiles are added). All values are given in seconds unless otherwise noted.

		CAS_1				CAS_10				CAS_100				CAS_1000			
		Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max
leftm.-up	1	23	0.11	0.02/0.03/0.05	13	160	0.77	0.02/0.03/0.06	127	32.4 min	9.39	0.03/0.04/0.06	28 min	4.2 h	73	0.02/0.04/0.05	3.4 h
	2	18	0.09	0.02/0.08/0.14	0.37	19	0.09	0.02/0.09/0.15	0.38	23	0.11	0.02/0.10/0.17	0.49	18	0.09	0.02/0.08/0.14	0.35
	total	**41**	0.2	0.05/0.13/0.18	13	**179**	0.86	0.05/0.13/0.19	127	**33 min**	9.5	0.06/0.15/0.23	28 min	**4.2 h**	73	0.05/0.12/0.17	3.4 h
leftm.-down	1	9	0.04	0.02/0.04/0.05	0.55	13	0.07	0.02/0.04/0.06	5.25	87	0.42	0.03/0.05/0.08	75	517	2.5	0.02/0.03/0.06	509
	2	18	0.09	0.02/0.08/0.13	0.35	18	0.09	0.02/0.08/0.13	0.34	24	0.12	0.03/0.11/0.19	0.47	17	0.08	0.02/0.08/0.13	0.34
	total	**27**	0.13	0.05/0.13/0.18	0.63	**31**	0.16	0.06/0.13/0.18	5.33	**111**	0.54	0.08/0.17/0.25	75	**534**	2.58	0.05/0.12/0.17	509
ff-up	1	31	0.15	0.11/0.17/0.22	0.31	32	0.15	0.11/0.18/0.23	0.31	45	0.22	0.15/0.24/0.31	0.43	31	0.15	0.11/0.17/0.22	0.3
	2	18	0.09	0.02/0.08/0.14	0.46	18	0.09	0.02/0.08/0.14	0.45	25	0.12	0.02/0.11/0.19	0.75	18	0.09	0.02/0.08/0.14	0.44
	total	**49**	0.24	0.14/0.26/0.33	0.68	**50**	0.24	0.14/0.26/0.33	0.68	**70**	0.34	0.19/0.35/0.47	1.1	**49**	0.24	0.13/0.25/0.33	0.67
ff-down	1	38	0.18	0.13/0.22/0.26	0.35	38	0.18	0.14/0.22/0.26	0.37	54	0.26	0.18/0.30/0.37	0.53	37	0.18	0.13/0.22/0.26	0.36
	2	19	0.09	0.02/0.08/0.13	0.38	19	0.09	0.02/0.08/0.14	0.37	26	0.13	0.03/0.11/0.19	0.51	19	0.09	0.02/0.08/0.13	0.35
	total	**57**	0.27	0.17/0.30/0.38	0.56	**57**	0.27	0.17/0.30/0.38	0.61	**80**	0.39	0.23/0.41/0.52	0.89	**56**	0.27	0.16/0.30/0.37	0.56
ffc-up	1	24	0.12	0.08/0.12/0.17	0.32	23	0.11	0.08/0.12/0.16	0.26	32	0.16	0.11/0.16/0.22	0.33	22	0.11	0.08/0.12/0.16	0.26
	2	19	0.09	0.02/0.08/0.15	0.49	19	0.09	0.02/0.08/0.14	0.47	26	0.12	0.02/0.11/0.19	0.62	18	0.09	0.02/0.08/0.14	0.46
	total	**43**	0.21	0.12/0.21/0.28	0.66	**42**	0.2	0.12/0.22/0.28	0.65	**58**	0.28	0.17/0.28/0.38	0.84	**40**	0.2	0.13/0.20/0.27	0.62
ffc-down	1	25	0.12	0.09/0.14/0.18	0.26	26	0.12	0.09/0.14/0.18	0.27	35	0.17	0.12/0.18/0.24	0.37	24	0.12	0.08/0.13/0.17	0.26
	2	20	0.09	0.02/0.09/0.14	0.38	20	0.1	0.02/0.09/0.14	0.56	27	0.13	0.03/0.12/0.19	0.6	18	0.09	0.02/0.08/0.13	0.35
	total	**45**	0.21	0.13/0.22/0.30	0.48	**46**	0.22	0.14/0.23/0.30	0.58	**62**	0.3	0.18/0.31/0.40	0.76	**42**	0.21	0.13/0.21/0.29	0.46

Table B.2: Extended version of Table 7.1 (values for quartiles are added). All values are given in seconds unless otherwise noted.

		CAS_1				CAS_10				CAS_100				CAS_1000			
		Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max
leftm.-up	1	0.54	0	0/0/0	0.08	0.57	0	0/0/0	0.07	0.48	0	0/0/0	0.07	0.51	0	0/0/0	0.08
	total	**20**	0.1	0.02/0.09/0.15	1.25	**20**	0.1	0.02/0.08/0.15	1.23	**20**	0.1	0.02/0.08/0.14	1.24	**20**	0.1	0.02/0.09/0.15	1.22
leftm.-down	1	0.57	0	0/0/0	0.08	0.52	0	0/0/0	0.08	0.54	0	0/0/0	0.09	0.6	0	0/0/0	0.08
	total	**19**	0.09	0.02/0.08/0.14	0.39	**20**	0.1	0.02/0.09/0.15	0.4	**20**	0.1	0.02/0.09/0.15	0.47	**20**	0.09	0.02/0.09/0.14	0.4
ff-up	1	0.7	0	0/0/0	0.26	0.69	0	0/0/0	0.24	0.85	0	0/0/0	0.33	0.86	0	0/0/0.01	0.26
	total	**19**	0.09	0.02/0.08/0.14	0.48	**19**	0.09	0.02/0.08/0.14	0.47	**23**	0.11	0.02/0.09/0.16	0.49	**23**	0.11	0.02/0.10/0.16	0.59
ff-down	1	0.8	0	0/0/0	0.31	0.81	0	0/0/0	0.28	0.9	0	0/0/0	0.31	0.96	0	0/0/0.01	0.29
	total	**20**	0.09	0.02/0.09/0.14	0.38	**19**	0.09	0.02/0.09/0.13	0.37	**23**	0.11	0.02/0.10/0.17	0.45	**22**	0.11	0.03/0.10/0.15	0.45
ffc-up	1	0.69	0	0/0/0	0.19	0.56	0	0/0/0	0.18	0.7	0	0/0/0	0.21	0.65	0	0/0/0	0.21
	total	**19**	0.09	0.02/0.08/0.14	0.48	**19**	0.09	0.02/0.08/0.13	0.46	**23**	0.11	0.02/0.10/0.17	0.49	**20**	0.1	0.02/0.08/0.15	0.47
ffc-down	1	0.71	0	0/0/0	0.21	0.55	0	0/0/0	0.2	0.63	0	0/0/0	0.21	0.56	0	0/0/0	0.19
	total	**20**	0.1	0.02/0.09/0.15	0.37	**19**	0.09	0.02/0.09/0.14	0.36	**21**	0.1	0.02/0.09/0.16	0.37	**19**	0.09	0.02/0.09/0.13	0.37

Table B.3: Extended version of Table 7.2 (values for quartiles are added). All values are given in seconds.

Algorithm 7.1		semantic		syntactic			
		Σ	mutants	Σ	ϕ	$Q_1/Q_2/Q_3$	max
leftm.-up	1	$>$ 5.98 h		23.8	0.04	0.02/0.02/0.03	3.6
	2	$>$ 41		2.2 h	11.5	0.16/2.20/19.82	52.9
	total	$>$ **6 h**	5/672	**2.2 h**	11.6	0.19/2.23/19.84	53.0
leftm.-down	1	$>$ 5.99 h		18.8	0.03	0.02/0.02/0.02	2.9
	2	$>$ 1		1.8 h	9.6	0.14/1.25/13.48	44.1
	total	$>$ **6 h**	4/672	**1.8 h**	9.6	0.16/1.28/13.49	44.1
ff-up	1	$>$ 5.98 h		17.0	0.03	0.02/0.02/0.02	0.6
	2	$>$ 87		1.9 h	10.2	0.14/1.72/17.15	39.4
	total	$>$ **6 h**	7/672	**1.9 h**	10.2	0.16/1.74/17.16	39.4
ff-down	1	$>$ 5.96 h		16.0	0.02	0.02/0.02/0.02	0.29
	2	$>$ 2.4 min		2.2 h	11.9	0.16/1.48/18.58	47.2
	total	$>$ **6 h**	8/672	**2.2 h**	11.9	0.18/1.50/18.60	47.2
ffc-up	1	$>$ 5.97 h		16.9	0.03	0.02/0.02/0.02	0.6
	2	$>$ 1.4 min		1.9 h	10.2	0.14/1.72/17.13	39.4
	total	$>$ **6 h**	6/672	**1.9 h**	10.2	0.16/1.74/17.15	39.4
ffc-down	1	$>$ 5.97 h		15.8	0.02	0.01/0.02/0.02	0.3
	2	$>$ 1.6 min		2.2 h	11.8	0.16/1.47/18.50	46.9
	total	$>$ **6 h**	7/672	**2.2 h**	11.8	0.18/1.49/18.51	46.9

Table B.4: Extended version of Table 7.4 (values for quartiles are added). All values are given in seconds unless otherwise noted.

		ffc-up				*leftmost-down*			
		Σ	ϕ	$Q_1/Q_2/Q_3$	max	Σ	ϕ	$Q_1/Q_2/Q_3$	max
Algorithm 7.1	total	**1.9 h**	10.2	0.16/1.74/17.15	39.4	**1.8 h**	9.6	0.16/1.28/13.49	44.1
	reach	51.4	0.07	-	-	52.7	0.08	-	-
Algorithm 7.2	find unsafe	44.1 min	3.9	0.06/0.74/6.75	17.0	42.8 min	3.8	0.06/0.58/5.52	17.4
	total	**45.1 min**	4.0	-	-	**43.8 min**	4.0	-	-
Algorithm 7.4	total	**32.1 min**	2.9	0.03/0.31/3.86	18.3	**33.2 min**	3.0	0.04/0.30/3.15	17.3
	reach	22.4	0.03	-	-	27.01	0.04	-	-
Algorithm 7.3	find unsafe	1.6 min	0.2	0.02/0.03/0.18	2.23	1.7 min	0.2	0.02/0.04/0.17	1.5
	total	**2.0 min**	0.2	-	-	**2.2 min**	0.2	-	-

Table B.5: Extended version of Table 7.5 (values for quartiles are added). All values are given in seconds unless otherwise noted.