Martin Tappler, BSc.[1]

Symbolic Input Output Conformance Checking of Action System Models

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Institute for Software Technology (IST)

Graz, December 18, 2015

[1] E-mail: martin.tappler@student.tugraz.at

Martin Tappler, BSc.[1]

Symbolische Überprüfung von “Input-Output Conformance” von “Action System”-Modellen

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium: Informatik

eingereicht an der

Technischen Universität Graz

Betreuer
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Institut für Softwaretechnologie (IST)

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: martin.tappler@student.tugraz.at

Abstract

This thesis uses actions systems as test models. It presents a symbolic execution-based approach to Input Output Conformance checking of action systems and discusses its application. The main focus lies on model-based mutation testing, which is a test-case generation technique. In principle, it generates distinguishing test cases from a specification model and mutated versions thereof. These mutated models are created by inserting faults into the original specification, thus test cases should cover such faults. Test cases are generated by applying a conformance check between models and transforming counter examples to conformance. This conformance check, however, is a performance bottleneck. Consequently, the main goal of this thesis is the development of a conformance check, which is efficient in terms of runtime.

Efficiency is achieved by symbolic handling of data to overcome the state-space explosion problem and adopting several optimisations. The main part of this thesis is formed by the formal presentation of the basic conformance checking algorithm, optimisations of this algorithm and relevant theoretical concepts. It further builds the basis for the implementation of a mutation-based test case generator, which is discussed as well.

In order to determine whether the symbolic execution-based approach pays off, several case studies involving the test case generator have been carried out and are analysed. They show that the conformance checker is indeed efficient, especially considering a comparison with a non-symbolic Input Output Conformance checker.

Since the conducted case studies show that the approach followed in this thesis is worthwhile, its applicability for further areas such as conformance checking of real-time system models is investigated.

Keywords: Input Output Conformance, ioco, Action Systems, Symbolic Execution, Model-Based Testing, Model-Based Mutation Testing, Conformance Checking, Test Case Generation.

Kurzfassung

Diese Arbeit verwendet “Action System”-Modelle als Testmodelle. Sie behandelt einen auf symbolischer Ausführung basierenden Ansatz zur Überprüfung von “Input Output Conformance” von “Action System”-Modellen. Des Weiteren wird seine Anwendung diskutiert, wobei der Fokus auf dem Hauptanwendungsgebiet, dem modellbasierten Mutationstesten, liegt, einer Technik zur Testfallgenerierung. Grundsätzlich werden dabei Testfälle generiert, die Unterschiede zwischen einem Spezifikations-Modell und mutierten Versionen davon erkennen. Diese mutierten Modelle werden duch das gezielte Einfügen von Fehlern in die ursprüngliche Spezifikation erzeugt, wodurch die erstellten Testfälle solche Fehler abdecken sollen. Durch Anwendung einer “Conformance”-Überprüfung und weitere Transformation der Gegenbeispiele für “Conformance” werden Testfälle erzeugt. Diese Überprüfung der “Conformance” zwischen Modellen ist allerdings rechenintensiv und kann die Anwendbarkeit der Technik beeinflussen. Aufgrund dessen ist das Hauptziel dieser Arbeit die Entwicklung einer effizienten Überprüfung von “Conformance”, wobei das Hauptaugenmerk auf Laufzeit gelegt wird.

Effizienz wird durch die symbolische Verarbeitung von Daten und durch den Einsatz verschiedener Optimierungen erreicht. Ersteres soll sicherstellen, dass das “State Space Explosion”-Problem nicht auftritt. Die formalen Beschreibungen der Grundversion des Algorithmus zur “Conformance”-Überprüfung, der Optimierungen dieses Algorithmus und relevanter Konzepte stellen den Hauptteil dieser Arbeit dar. Diese Beschreibungen bilden weiters die Basis für die Implementierung eines mutationsbasierten Testfallgenerators, welcher ebenfalls diskutiert wird.

Um festzustellen, ob sich der vorgestellte Ansatz als sinnvoll erweist, wurden mehrere Fallstudien durchgeführt und analysiert. Diese Fallstudien haben gezeigt, dass die implementierte “Conformance”-Überprüfung effizient arbeitet. Das wird vor allem bei der Betrachtung eines Vergleichs mit einer nicht-symbolischen “Input Output Conformance”-Überprüfung offensichtlich.

Da die durchgeführten Fallstudien gezeigt haben, dass der in dieser Arbeit verfolgte Ansatz lohnenswert ist, wird seine Anwendbarkeit für weitere Gebiete untersucht. Ein Beispiel für ein mögliches Gebiet, ist die “Conformance”-Überprüfung von Modellen von Echtzeit-Systemen.

Schlagworte: Modellbasiertes Testen, Modellbasiertes Mutationstesten, symbolische Ausführung, “Conformance”-Verifikation, Testfallgenerierung, Action Systems, “Input Output Conformance”, ioco.

Affidavit

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

. . . . .	. . . . .
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. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

. . . . .	. . . . .
Ort, Datum	(Unterschrift)

Acknowledgements

I would like to thank all the people who supported me during my studies and especially during the writing of this thesis. I would like to address special thanks to my advisor Bernhard K. Aichernig who sparked my interest in the area of software verification and introduced me to formal approaches to software developement, which led to the creation of this thesis. During this process, he was always open to discuss ideas and problems and thereby shaped the work presented in the following.

Furthermore, I want to thank my colleagues and fellow students who helped during this project: Florian Lorber for his help regarding modelling of real-time systems, Benedikt Maderbacher for his support during the early phase of the implementation and Severin Kann for providing me with Event-B models.

I also would like to thank my partner Nina for her support and for keeping my motivation up. Last but not least, I am grateful to my family, especially to my parents Helga and Gerhard, who supported and encouraged me throughout my studies.

This work has been financially supported by the Austrian Research Promotion Agency (FFG), project number 845582, Trust via cost function driven model based test case generation for non-functional properties of systems of systems (TRUCONF).

Martin Tappler
Graz, Austria, December 18, 2015

Danksagung

Ich möchte mich herzlich bei allen bedanken, die mich im Laufe meines Studiums und im Speziellen während des Verfassens dieser Arbeit unterstützt haben. Besonderer Dank gilt meinem Betreuer Bernhard K. Aichernig, der mein Interesse an Softwareverifikation geweckt und mich auf formale Ansätze zur Softwareentwicklung aufmerksam gemacht hat. Das hat zur Erstellung dieser Arbeit geführt, während der er immer offen für Diskussionen von Problemen und Ideen war. Dadurch hat er bedeutenden Einfluss auf die Arbeit genommen.

Des Weiteren möchte ich meinen Kollegen und Studienkollegen danken, die mir während dieses Projekts geholfen haben: Ich danke Benedikt Maderbacher für seine Unterstützung während der Frühphase der Implementierung, Florian Lorber für seine Hilfe bezüglich der Modellierung von Echtzeit-Systemen und Severin Kann für die Bereitstellung von Event-B-Modellen.

Ich möchte auch meiner Partnerin Nina für ihre Unterstützung danken und dafür, dass sie mich fortwährend motiviert. Zu guter Letzt möchte ich meiner Familie Dank aussprechen, allen voran meinen Eltern Helga und Gerhard, die mich während meines Studiums unterstützt und ermutigt haben.

Diese Arbeit wurde finanziell unterstützt von der Österreichischen Forschungsförderungsgesellschaft (FFG) im Rahmen des Projekts TRUCONF mit der Projektnummer 845582, Trust via cost function driven model based test case generation for non-functional properties of systems of systems.

Martin Tappler
Graz, Österreich, 18. Dezember 2015

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Acronyms

ioco Input Output Conformance.

sioco Symbolic Input Output Conformance.

stioco Symbolic Timed Input Output Conformance.

tioco Timed Input Output Conformance.

API Application Programming Interface.

AST Abstract Syntax Tree.

BNF Backus-Naur Form.

DQTS Divergent Quiescent Transition System.

IOLTS Input Output Labelled Transition System.

IOSTS Input Output Symbolic Transition System.

IOTS Input Output Transition System.

JVM Java Virtual Machine.

LTS Labelled Transition System.

SMT Satisfiability Modulo Theories.

STS Symbolic Transition System.

SUT System Under Test.

UML Unified Modeling Language.

VDM Vienna Development Method.

1 Introduction

1.1 Motivation

The verification of software is an important task in the software development life cycle. This becomes apparent when looking at the growing influence of electronic devices on our lives and the possible impact of software failures. The consequences of defects in software systems range from the loss of a company's reputation to threats to lives of human beings [66]. Through the rising complexity of software systems, the verification task becomes more complex as well, which further adds to the importance of research and education in this area.

There exist two main approaches to ensure that a system meets its requirements. In the first one, software testing, several tests are carried out to check if the System Under Test (SUT) works correctly. In the second one, mathematical proofs are used to show that a system is correct with respect to the specification [19, 84]. Since it is difficult to perform proofs for industrial systems, software testing is predominantly used for quality assurance in practice today.

Although simpler in general, software testing is a labour intensive task as it requires an engineer to select relevant scenarios and conditions, which are used to test the SUT. Consequently traditional software testing is also error-prone. Hence, there is a pressing need to provide assistance in the testing process through the automation of steps involved in this process. A common solution to this problem advocates the adoption of an abstract model of the SUT, from which test cases can be generated automatically based on some criterion. This approach is called model-based testing and will be discussed in the next section.

This thesis focuses on a specific form of model-based testing which uses mutation. It builds upon existing research performed in the area of model-based mutation testing [3, 58] and evaluates an approach to test case selection based on symbolic execution and the Input Output Conformance (**ioco**) relation [78].

1.2 Model-Based Testing

As pointed out above, in model-based testing an abstract model of the SUT is used to specify the intended behaviour of the SUT. Possible sources of abstraction are for instance the omission of functionality or the simplification of data [80]. Generally, software tests need to specify a number of execution scenarios and the properties which need to be satisfied for a test to be successful. Model-based testing provides solutions to both of these problems [80]. As the model represents a formalisation of a set of requirements imposed on the SUT, it acts as a test oracle. Hence, it can provide answers as to whether the outputs of the SUT conform to the specification given by the requirements. Furthermore, the model together with a test selection criterion can be used to select and generate a set of test cases.

Figure 1.1 shows the model-based testing process. It can be divided into five steps [80]:

1. Creation of a model from the requirements given in natural language
2. Definition of a test case selection criterion
3. Translation of selection criteria into test case specifications
4. Generation of test cases from the model and a test case specification
5. Execution of test cases on the SUT, whereby an adaptor is used to help bridge the abstraction gap between model and SUT

The result of the last step is a set of verdicts, one for each test case. A verdict may either be *pass*, if the SUT has shown conforming behaviour, *fail*, if the SUT has shown non-conforming behaviour or

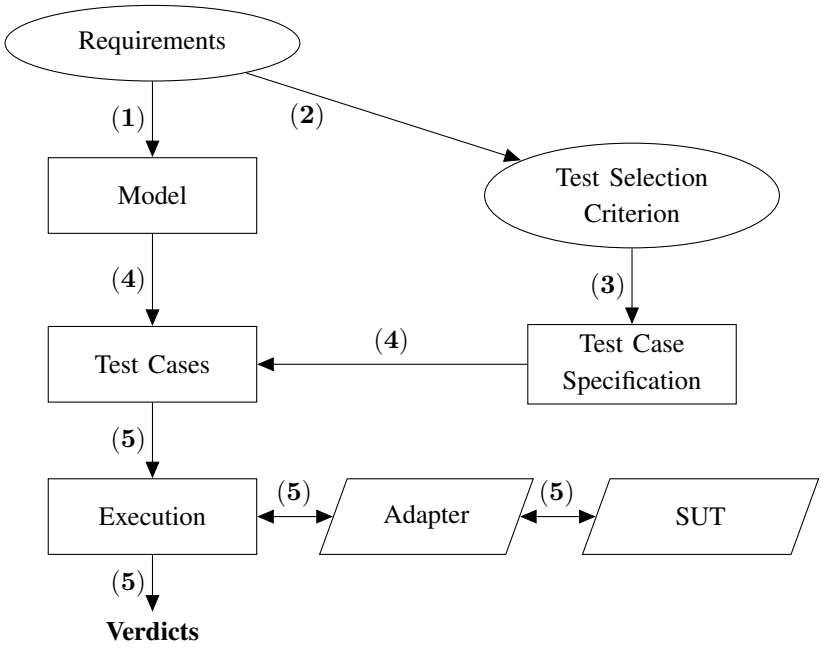


Figure 1.1: The model-based testing process adapted from Utting et al. [80]. The arrow labels correspond to the steps in the process.

inconclusive. The last verdict is assigned if it is not possible to decide whether the output of the SUT was correct or if the goal of the test was not achieved.

1.3 Conformance Testing

In the last section, verdicts and a decision criterion for assigning verdicts have been introduced. The decision criterion, however, distinguishes between conforming and non-conforming behaviour while the conditions necessary for conformance are yet to be discussed. Unfortunately, a general answer to the question "When does an SUT conform to its specification?" cannot be given. Nevertheless, it is possible to discuss important aspects and assumptions in the context of conformance testing.

The specification model must be given in a formalism with precise formal semantics. Tretmans for instance considers formalisms with semantics which can be expressed in terms of Labelled Transition Systems(LTSs) [77]. As is typical in model-based testing, implementations are considered to be black boxes, that is, their internal structure is not known. In order to be able to formally reason about conformance an assumption has to be made, which is referred to as the test hypothesis [18, 77]: a formal model capturing the behaviour of the implementation is assumed to exist. It must be possible to express this model in a way such that it is possible to reason about it formally.

Based on this assumption, it is possible to give a condition for conformance. Consider a formalism, which is able to express models belonging to some set $\mathcal{F}$: following [77], a conformance relation $\mathbf{imp} \subseteq \mathcal{F} \times \mathcal{F}$ should be used to decide whether an implementation $i \in \mathcal{F}$ conforms to a specification $s \in \mathcal{F}$. Conformance of i to s is denoted by $i\ \mathbf{imp}\ s$, which can be interpreted as i implements the specification s. Generally, implementations and specifications may also belong to different sets $\mathcal{F}_I$ and $\mathcal{F}_S$, thus $\mathbf{imp} \subseteq \mathcal{F}_I \times \mathcal{F}_S$.

Hence, the question of conformance between a specification and an implementation can only be answered relative to a conformance relation. While this provides the freedom of defining arbitrary conditions for conformance, there exists a variety of well-known and studied conformance relations. Essentially, this adds another dimension to the taxonomy defined by Utting et al. [80] with interdependencies with other dimensions.

Prominent examples of conformance relations are for instance observational equivalence and strong and weak bisimulation equivalence [77]. However, equivalence relations may be too restrictive [2]. Since models are incomplete in most cases, conformance relations should account for implementation freedom for unspecified details. More suitable choices are order relations like refinement which is used in [9]. Another popular conformance relation is **ioco** [78], which essentially requires that an implementation must only produce outputs which are allowed by the specification. It allows for the usage of non-deterministic models and provides implementation freedom for unspecified inputs. For these reasons, a symbolic variant of it will be used as conformance relation in this thesis. Section 2.2 will introduce **ioco** and associated concepts formally.

1.4 Model-Based Mutation Testing

Up to now, the model-based testing process and conformance, on which criteria for test success can be based, have been discussed. In the following, an approach to test case generation shall be discussed.

Mutation testing was initially introduced to assess the adequacy of a test data set for some program [40]. The basic working principle is as follows:

1. The program is executed using the given test data. If this reveals an error, the program is incorrect, otherwise the actual mutation testing can be performed.
2. During program mutation, simple errors, which are also called mutations, are seeded into the program creating a number of faulty implementations of the program. These faulty mutations are also referred to as mutants.
3. In the last step, the mutants are tested using the given data. If all mutants are detected to be incorrect, the test data is deemed adequate, otherwise it should be extended.

The types of errors introduced into the program are governed through transformation rules also called mutation operators [57].

It should be noted that mutants may also be equivalent to the original program. This may happen if mutations for instance affect unreachable code. Hence, the test data should identify all non-equivalent mutants as incorrect. The mutation score, which is the ratio between the number of detected mutants and the number of non-equivalent mutants, is generally used to measure the effectiveness of test sets [57].

In the following, important aspects of mutation testing shall be discussed. Mutation testing relies on two assumptions:

- Programmers develop programs that differ from the correct version only by small deviations. This is also referred to as the competent programmer hypothesis [57].
- If test data can detect all kinds of simple errors, then it is so sensitive that it can also detect more complex errors. This is also referred to as the coupling effect [40].

Finally, the notion of simple errors shall be concretised. A simple error is some error which commonly occurs in programming [40]. This may for instance be the usage of wrong constant values or relational operators. Furthermore, a simple error affects the implementation only at one point. Program mutation therefore changes only one statement of a program for instance to create one mutant.

More generally, mutant types can be distinguished based on the number of errors they contain [57]. Mutants containing only one simple error are referred to as first-order mutants, while mutants containing more than one error are called higher-order mutants.

The concept of model-based mutation testing shall now be introduced. While it shares similarities, for instance in terms of terminology, with traditional mutation testing, its purpose is a different one. Roughly speaking, model-based mutation testing is a specific form of model-based testing which uses a

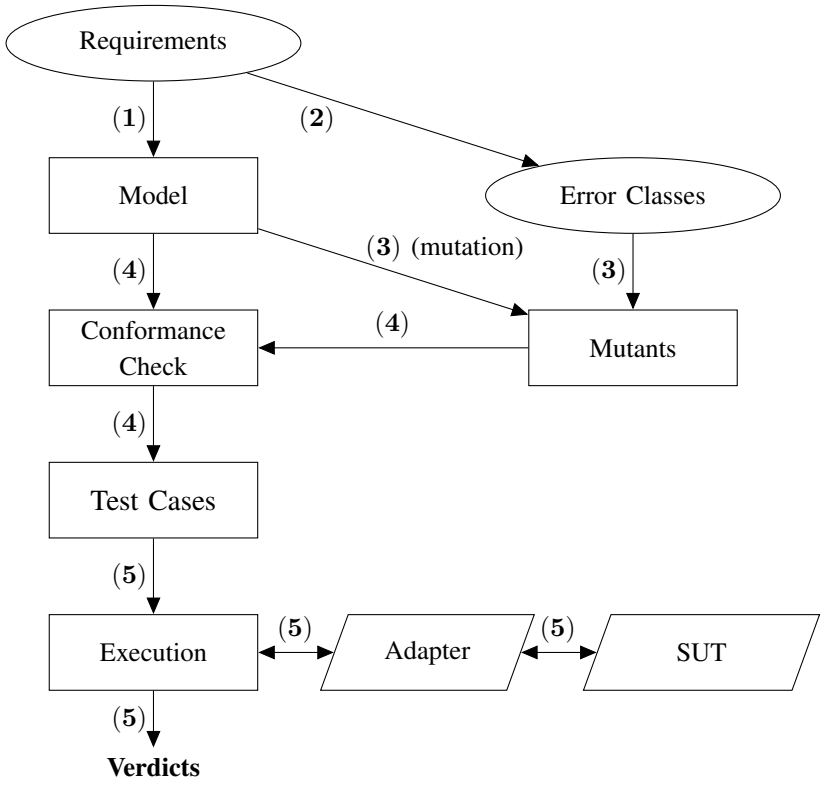


Figure 1.2: The model-based mutation testing process adapted from Aichernig et al. [9]. The arrow labels correspond to the steps in the model-based testing process.

fault-based test case selection criterion in the second step of the process discussed in Section 1.2. Hence, model-based mutation testing affects steps two to four in the aforementioned process. Their specific form in this context is discussed in the following. The discussion is based on the process description given by Aichernig et al. [9]. A visual representation of the model-based mutation testing process is given in Figure 1.2.

In model-based mutation testing, an informal test case selection criterion can for instance be given as *"The test cases should cover all possible errors from X"*, where X is a set of error classes. The translation of this criterion into formal test case specifications is done via mutation of the model of the SUT, whereby X defines the set of mutation operators to be applied. Informally speaking, the generated test cases should cover the injected faults. Hence, each non-equivalent mutant m corresponds to a test case specification. The specified test cases should reveal non-conforming behaviour of programs implementing the faulty model m. In order to generate test cases a search for conformance violations with respect to the original model is performed for every created mutant. This process generates test cases for all conformance violations that are found.

It can be seen that conformance testing is not only relevant for assigning test verdicts but also for distinguishing the original specification model from a mutant model. For this purpose, a mutant is interpreted as an implementation and conformance between mutant and specification is checked. This is possible because both specification and mutant are defined using the same formalism, about which it is possible to reason.

However, differently from the situation described in Section 1.3, the testing can actually be performed as white-box testing, that is, the structure of the mutant can be utilised. Since the equivalence of a mutant to a specification is defined with respect to a conformance relation, equivalent model mutants will also be referred to as *conforming mutants* in the remainder of this thesis. Conformance testing on model-level is actually performed exhaustively up to a given bound in many cases, which means that the entire system behaviour below the bound is covered. Consequently, it is also referred to as conformance checking.

This applies to this thesis as well, thus the terms conformance testing on model-level and conformance checking will be used synonymously. However, the latter term will be preferred. It is actually used, because conformance checking is a specific form of model-checking [33]. But unlike traditional model-checking which checks whether a system description satisfies a property expressed in temporal logic, conformance checking checks a different kind of property. Given an implementation, it checks whether it conforms to a specification.

The intuition behind the model-based mutation testing approach is that a test case generated for some mutant m would fail if it was executed on an SUT which implements m and thus contains the same error as m. Hence, using model-based mutation testing it is possible to cover a range of error classes via the choice of suitable mutation operators. Aichernig et al. have shown that this technique is indeed effective [4], as it is possible to achieve a high mutation score. Furthermore, evidence has been given that supports the assumption that it also benefits from the coupling effect.

1.5 Stepwise Development of Test Models

Generally, a model used for model-based testing should be abstract in order to be able to reason about its validity [80], yet models should incorporate all important aspects of the system. Formal methods such as the Vienna Development Method (VDM) [59], the RAISE method [49] and Event-B [1] provide a means of coping with this problem. Using these methods it is possible to start development with an abstract model which fulfils some desired properties and to create a series of models with increasing complexity. In order to show that concrete models fulfil properties defined by more abstract models, conformance checks are necessary between any two consecutive models. In other words, it must be shown that a concrete model conforms to the abstract model from which it was derived. It should be noted that conformance is also referred to as refinement in formal methods and an implementation of a model M is said to be a refinement of M.

In general, conformance/refinement is checked through manual proofs. However, alternatively also other tools like model-checkers [50, 69] or Satisfiability Modulo Theories (SMT)-solvers [39] may be used to validate properties expressed in formal methods. Hence, conformance testing on model-level may also be useful during the development of test models. It opens up the possibility to start with a simple model, validate this model and to add details in a stepwise manner. Automatic conformance checks performed after each step would ensure that concrete models do not violate requirements expressed in more abstract models.

This approach to development is actually facilitated by the conformance relation **ioco**, which is considered in this thesis. It allows implementation freedom for non-specified inputs [78]. So it is possible to model behaviour corresponding to a small set of inputs in the most abstract model and to add behaviour corresponding to further inputs in more concrete models. Finally, when the full set of requirements has been formalised through modelling, test cases can be generated from the most concrete model. This process is outlined in Figure 1.3.

1.6 Problem Statement and Goals

As pointed out before, model-based mutation testing is an effective approach to software testing. However, test case generation in this context involves conformance checks for all generated mutants, which may be very expensive in terms of computation time. The excessive amount of time needed to check conformance may even render the approach inapplicable in some cases. This problem is for instance discussed by Aichernig et al. [9][1]. A number of factors influence the computation time including the size

[1]The tool Ulysses checks conformance in a concrete way and cannot process the models CAS_100 and CAS_1000 in reasonable time.

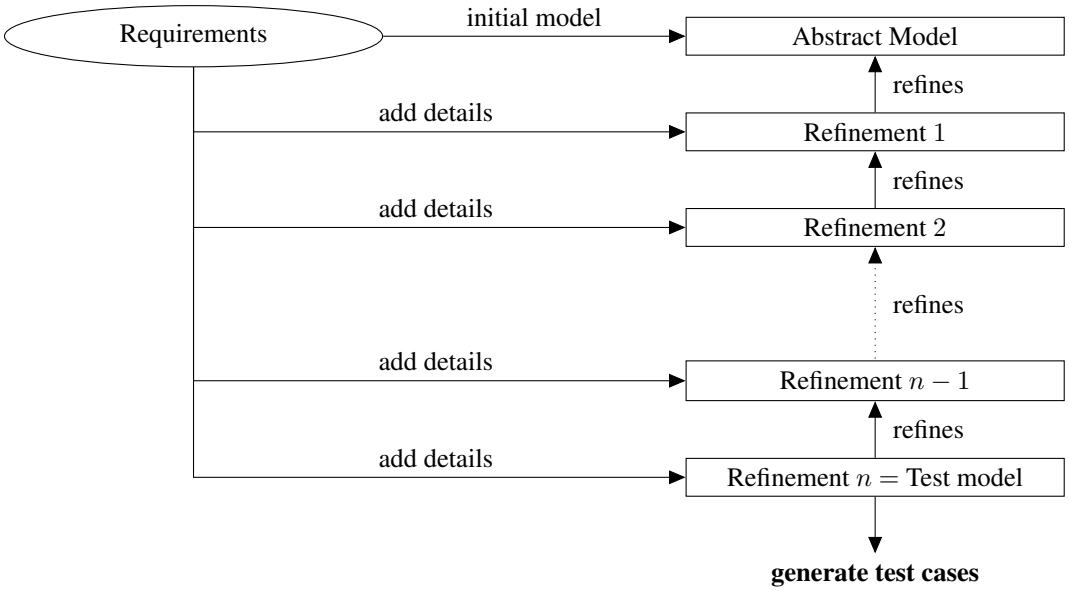


Figure 1.3: Stepwise development of test models by adding details in n steps. Based on the last refinement, test cases can be generated.

of the model, the type and number of mutation operators, the conformance relation and of course the implementation of the conformance check.

The implementation of the **ioco** conformance check used in Chapter 10 of Elisabeth Jöbstl's dissertation [58] and the corresponding experiments can be seen as the predecessor project of this thesis. These tasks were carried out in joint work performed by Elisabeth Jöbstl and the author of this thesis. The experiments revealed that the **ioco** conformance check faces efficiency problems.

Consequently, the approach presented in the following aims at improving efficiency while focusing on the same kind of models, thus models of reactive systems will be targeted. Reactive systems are systems which continuously respond to their environment upon receiving inputs [52]. They do not strictly wait to receive inputs in order to perform calculations and output the results as opposed to transformational systems. Such systems rather provide outputs depending on inputs and time in an ongoing relationship with users.

The focus lies on the same kind of systems to be able to compare both approaches. Additionally, a similar modelling formalism suited for reactive systems shall be used. More concretely, a simplified form of the action system language used in [58] will serve as the modelling formalism. Furthermore, the conformance check shall be based on **ioco** as well.

Since the previous implementation of the **ioco** conformance check follows an enumerative approach, it shows performance issues especially when models with large state space or large number of executable paths are checked. In other words, it suffered from the state explosion problem [34]. Hence, the conformance check approach followed in this thesis shall avoid explicit enumeration. This can be achieved by symbolic handling of data.

This leads to the main goal of this thesis: in order to overcome the state space explosion problem, a symbolic **ioco** conformance check shall be developed. The development will be based on the Symbolic Input Output Conformance (**sioco**) relation developed by Frantzen et. al [45] and follows a white-box testing approach. As the efficiency of the conformance check was identified to be of utmost importance, the **sioco** conformance check shall be implemented as efficiently as possible.

Another goal of this thesis is to evaluate this approach and to compare its efficiency to the implementation of the concrete **ioco** conformance checker. For this purpose, it is necessary to implement a model mutation system and to perform runtime measurements using both implementations.

Furthermore, to demonstrate the applicability of the approach, symbolic test cases shall be generated and a test driver to execute those tests shall be developed.

To summarise, it shall be possible to efficiently check **ioco** conformance between two action system models, which supports two use cases:

- stepwise development of test models
- model-based mutation testing

For the latter, all components necessary to actually perform tests on an SUT shall be implemented as well. Unless otherwise noted, the focus lies on efficiency in terms of computation time.

1.7 Published Material

Prior to the completion of this thesis, a paper has been written in joint work with Bernhard Aichernig. It covers the most important developments of this thesis and is fifteen pages long. It has been submitted for presentation at the 1^{st} *Usages of Symbolic Execution Workshop* (USE) co-located with the 20^{th} *International Symposium on Formal Methods* held at the Department of Informatics of the University of Oslo. After it had been peer-reviewed, it was accepted for presentation at the workshop, which was subsequently given by the author of this thesis [12]. The workshop proceedings will be published within the *Electronic Notes in Theoretical Computer Science*[2].

As a result, some concepts presented in the following have already been discussed in a very similar form before. These concepts include for instance the modelling formalism, the approach to conformance checking and a comparison with the previously implemented enumerative **ioco** checker mentioned above. However, this thesis covers theoretical foundations and optimisations in more depth. Additionally, it discusses further case studies and topics such as test case execution and implementation-specific details. Nevertheless, whenever a section covers published material, it will be indicated at the beginning of the section.

1.7.1 Related Publications

The author of this thesis has been involved in the creation of two additional publications related to the area of **ioco** checking, which shall be discussed shortly.

Does this Fault Lead to Failure? - Combining Refinement and Input-Output Conformance Checking in Fault-Oriented Test-Case Generation. This paper discusses the application of a combination of refinement and **ioco** [8] as conformance relation in model-based mutation testing. More concretely, a mutant is considered to be conforming if it refines the specification, otherwise **ioco** is checked to determine whether the mutant actually conforms to the specification. A test case is generated if an **ioco** conformance violation is found. Hence, test cases are only generated if an internal error detected by the stricter refinement check propagates to a visible failure detected by the **ioco** check. The rationale behind this approach is that performance, as compared to pure **ioco** checking, may be increased because refinement can be checked more efficiently than **ioco** and **ioco** need not be checked if the mutant refines the specification. Two case studies indicate that this assumption is valid.

Conformance Checking of Real-Time Models - Symbolic Execution vs. Bounded Model Checking. This paper focuses on conformance checking of real-time system models with the purpose of mutation-based test case generation [11]. It compares an existing approach based on bounded model-checking [10] to a newly developed approach based on symbolic execution with respect to runtime. The new approach

[2] http://www.entcs.org/ (last visit: 29.10.2015)

is actually an extension of the conformance checking approach followed in this thesis, but explicitly accounts for time. It will be discussed in more detail in Section 8.2.

1.8 Structure of this Thesis

This thesis is structured as follows:

Chapter 2 introduces concepts necessary for the development and discussion of the **sioco** conformance check. These concepts comprise notational conventions, the **ioco** conformance relation and LTSs, which are usually used to define **ioco**, the modelling formalism used in this thesis, symbolic execution and the **sioco** conformance relation. The technique of symbolic execution is a white-box testing technique based on symbolic computations. Building upon symbolic execution of action systems, **sioco** is defined.

Chapter 3 and Chapter 4 form the main part of this thesis, as they presents an abstract view of the **sioco** conformance checker. The view is abstract in the sense that it does not focus on a specific implementation technology. The chapters rather aim at providing general guidance for the implementation of such a conformance checker. Chapter 3 is split into two sections: the first section introduces further concepts involved in the development. These concepts are used in the second section of Chapter 3 and in Chapter 4, which present the conformance checking algorithm and optimisations thereof respectively.

Chapter 5 discusses the application of the **sioco** conformance check. More concretely, the application for model-based mutation testing and model-checking will be examined. Since the focus lies on testing, testing aspects will be covered more in-depth.

Chapter 6 highlights important implementation aspects of the mutation-based test case generator implemented in the course of this thesis.

Chapter 7 provides evidence for the efficiency of the presented approach by listing experimental results. In addition to listing measurement results, the results are discussed and compared to measurements performed with the explicit **ioco** checker used in [58]. Hence, the secondary goal of this thesis is addressed in this chapter, a comparison between the symbolic and the concrete approach to **ioco** conformance checking.

Chapter 8 presents further work building upon the implemented test case generator. It for instance covers extensions of the conformance checker, implemented to permit more convenient modelling. Another extension discussed in this chapter supports conformance checking of real-time system models.

Chapter 9 gives a summary and an overview of related research in the area of testing and conformance verification. A discussion of findings and future work concludes this thesis.

2 A Symbolic Framework for Conformance Checking

This chapter introduces concepts which form the basis for subsequent chapters. At the beginning, first-order logic concepts and notational conventions used for discussing theoretical aspects of this thesis will be presented. This is followed by definitions of LTSs and the **ioco** relation, which will not be used directly for the development of the conformance checker, but will serve as a context for definitions and examples. The syntax and semantics of action systems, the used modelling formalism, will also be introduced formally. A discussion of symbolic execution on both implementation-level and model-level concludes this chapter. In the context of symbolic execution on model-level, the **sioco** conformance relation will be defined.

The chapter title is derived from the title of the paper *"A Symbolic Framework for Model-Based Testing"* written by Frantzen et al. [45], on which this chapter is based.

2.1 First-order logic

A shortened version of this section introduces notational conventions in the USE-workshop paper [12].

The work by Frantzen et al. [45] forms the basis for the development of the conformance checker, as it defines the **sioco** conformance relation. This thesis will follow the same style in terms of formal derivations and definitions. As a result, the same concepts from first-order logic and notational conventions will be used. This section shall give a short overview of these, while closely following [45].

For two sets A and B the set of all total function from A to B will be denoted B^A. To compose two functions $f : A \to B$ and $g : B \to C$, the operator $\circ$ will be used, thus the composition of f and g will be denoted by $g \circ f$, which is a function mapping from A to C. If allowed by the context, a tuple $\langle x_1, \ldots, x_n \rangle$ may be interpreted as the set $\{x_1, \ldots, x_n\}$.

All formulas make use of first-order logic. Therefore, a first-order structure $(\mathfrak{S}, \mathfrak{M})$ will be assumed to exist, where $\mathfrak{S} = (F, P)$ is a single-sorted logical signature. It should be noted that the simplifying restriction to single-sorted structures does not limit applicability of the presented work, as many-sorted structures can be reduced to single-sorted structures [72]. The signature describes the syntactical elements of the structure, that is, it defines a set of function symbols F and a set of predicate symbols P. The model $\mathfrak{M}$ describes the semantical aspects of the structure, it is a tuple $(\mathfrak{U}, (f_{\mathfrak{M}})_{f \in F}, (p_{\mathfrak{M}})_{p \in P})$. The non-empty set $\mathfrak{U}$ is called universe and contains the values, variables and constants may take. All functions $f_{\mathfrak{M}}$ map from tuples of values in $\mathfrak{U}$ to single values in $\mathfrak{U}$ defining the semantics of function symbols f. Models for predicates $p \in P$ are given through $p_{\mathfrak{M}} \subseteq \mathfrak{U}^n$, where n is the arity associated with predicate symbol p.

For a set of variables X, the set of terms over X, which are built from $f \in F$ and $x \in X$, will be denoted by $\mathfrak{T}(X)$. The set of variables in a term t will be denoted by $\mathsf{var}(t)$.

First-order formulas are inductively defined as follows [54]:

- if t_1 and t_2 are terms, then $t_1 = t_2$ is a formula,
- if $p \in P$ is a predicate symbol with arity 0, then p is a formula (p is a "propositional atom")
- if $t_1, \ldots, t_n$ are terms and $p \in P$ is a predicate symbol with arity $n \geq 1$, then $p(t_1, \ldots, t_n)$ is a formula,
- if ϕ and γ are formulas, then $\phi \otimes \gamma$, with $\otimes \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, is a formula,
- if ϕ is a formula, then $\neg\phi$ is a formula,

- if ϕ is a formula and x is a variable, then $\gamma = \exists x : \phi$ and $\gamma = \forall x : \phi$ are formulas. The variable x is a bound variable in γ.

Note that this definition explicitly requires the existence of a special equality-predicate $=$ with arity two. Similarly to terms, the set of formulas with free variables in X is denoted by $\mathfrak{F}(X)$ and a function free is introduced, which maps a formula ϕ to the set of free variables in ϕ.

Let $\mathfrak{X}, X$ and Y be sets of variables, with $X, Y \subseteq \mathfrak{X}$. A term-mapping σ is a function mapping from variables to terms, that is, $\sigma : \mathfrak{X} \to \mathfrak{T}(\mathfrak{X})$, which is extended to tuples by $\sigma(\langle x_1, \ldots, x_n \rangle) = \langle \sigma(x_1), \ldots, \sigma(x_n) \rangle$. σ_X will be used to denote the term-mapping restricted to the variables in X, with

$$\sigma_X(x) = \begin{cases} \sigma(x) & \ldots x \in X \\ x & \ldots \text{otherwise.} \end{cases}$$

In the following $\mathfrak{T}(Y)^X \subseteq \mathfrak{T}(\mathfrak{X})^{\mathfrak{X}}$ will be used to denote the set of term-mappings mapping from variables in X to terms over Y. It holds that $\sigma(x) \in \mathfrak{T}(Y)$ for $x \in X$ and $\sigma(x) = x$ otherwise. Given a term-mapping σ, a substitution $t[\sigma]$ replaces the variables in a term t, which are given through $\mathsf{var}(t)$, by the mappings defined by σ. Analogously, $\phi[\sigma]$ replaces the free variables in a formula ϕ, which are given through $\mathsf{free}(\phi)$. Substitutions applied for formulas do not add bound variables through implicit proper renaming. It follows that $[\sigma] : \mathfrak{F}(\mathfrak{X}) \cup \mathfrak{T}(\mathfrak{X}) \to \mathfrak{F}(\mathfrak{X}) \cup \mathfrak{T}(\mathfrak{X})$.

For a formula ϕ the existential closure for X, denoted by $\overline{\exists}_X \phi$, is equivalent to ϕ but with variables in X existentially quantified, that is, $\overline{\exists}_X \phi = \exists x_1, \exists x_2, \ldots \exists x_n : \phi$ for $\{x_1, x_2, \ldots, x_n\} = X \cap \mathsf{free}(\phi)$. The universal closure, denoted by $\overline{\forall}_X \phi$, is defined likewise.

A valuation υ defines values for variables. Hence, υ is a function mapping variables in a set X to values in the universe $\mathfrak{U}$, thus it is an element of the set $\mathfrak{U}^X$. It is extended to tuples through $\upsilon(\langle x_1, \ldots, x_n \rangle) = \langle \upsilon(x_1), \ldots, \upsilon(x_n) \rangle$. Two valuations $\upsilon \in \mathfrak{U}^X$ and $\varsigma \in \mathfrak{U}^Y$ be combined through union to $\upsilon \cup \varsigma \in \mathfrak{U}^{X \cup Y}$ if $X \cap Y = \emptyset$. The union is defined by

$$(\upsilon \cup \varsigma)(x) = \begin{cases} \upsilon(x) & \ldots x \in X \\ \varsigma(x) & \ldots x \in Y. \end{cases}$$

Based on a valuation, a term can be evaluated: the values for variables are given through the valuation and the value of the term is calculated through the $f_{\mathfrak{M}}$ defined in the model. Given a valuation $\upsilon \in \mathfrak{U}^X$, a term-evaluation is denoted by $\upsilon_{\mathsf{eval}} \in \mathfrak{U}^{\mathfrak{T}(X)}$. In order to evaluate constant terms, that are terms which do not contain variables, the function $\mathsf{eval} \in \mathfrak{U}^{\mathfrak{T}(\emptyset)}$ is introduced. Finally, $\upsilon \models \phi$ shall denote the satisfaction of a formula ϕ with respect to a valuation υ. Hence, $\upsilon \models \phi$ expresses that ϕ is equivalent to $\top$, when the free variables in ϕ are replaced by the values given through υ and the resulting formula is interpreted based on the model $\mathfrak{M}$. The symbol $\top$ represents a tautology and the negation of it is given by $\neg\top = \bot$.

Example 2.1 (First-order Logic).
The application of the concepts presented above shall be demonstrated by means of various concrete examples. Let $(\mathfrak{S}, \mathfrak{M})$ be a first-order structure based on an integer universe, where:

- $\mathfrak{S} = (F, P)$ *with* $F = \{0, plus\}$, $P = \{gt\}$, $\mathsf{arity}(0) = 0$, $\mathsf{arity}(plus) = 2$ *and* $\mathsf{arity}(gt) = 2$
- $\mathfrak{M} = (\mathbb{Z}, \{0_{\mathfrak{M}}, plus_{\mathfrak{M}}\}, \{gt_{\mathfrak{M}}\})$ *with* $0_{\mathfrak{M}} = 0$, $plus_{\mathfrak{M}}((x, y)) = x + y$ *for* $x, y \in \mathbb{Z}$ *and* $gt_{\mathfrak{M}} = \{(x, y) \in \mathbb{Z}^2 | x > y\}$

Hence, this structure defines a function for adding integers and a predicate for comparing integers. Furthermore, it defines a function without parameters to represent the constant 0*. Given a set of variables* $A = \{x, y\}$, $t = plus(plus(x, y), 0)$ *is an example of a term with* $\mathsf{var}(t) = A$ *and* $\phi = gt(t, 0) \vee 0 = x$ *is a formula with* $\mathsf{free}(\phi) = A$*. However, the existential closure with respect to* A *does not contain free variables, that is,* $\mathsf{free}(\overline{\exists}_A \phi) = \{\}$.

Let $X = \{x\}$ and $Z = \{z\}$ be two singleton sets of variables and let $\sigma \in \mathfrak{T}(Z)^X$ with $\sigma(x) = plus(z, z)$ be a term-mapping: the substitutions for the term t and the formula ϕ are given by $t[\sigma] = plus(plus(plus(z, z), y), 0)$ and $\phi[\sigma] = gt(plus(plus(plus(z, z), y), 0), 0) \vee 0 = plus(z, z)$.

Valuations shall now be investigated: let A be defined as above and let $\upsilon \in \mathfrak{U}^A$ be a valuation with $\upsilon(y) = 2$ and $\upsilon(x) = 3$. A term-evaluation with respect to υ is given by $\upsilon_{\mathsf{eval}}(t) = (3 + 2) + 0 = 5$. The valuation satisfies ϕ, that is $\upsilon \models \phi$, because $5 > 0 \vee 0 = 3 \Leftrightarrow \top$.

Note that in the following, well-known predicates and functions will be written in infix notation on the syntax-level rather than in prefix notation to enhance readability.

2.2 Labelled Transition Systems and Input Output Conformance

In the following, a definition of LTSs will be given. It will be based on the definitions given in [78] and in [45]. The LTS formalism allows for modelling of systems by defining a set of states, an initial state and the transitions between those states. The transitions are labelled by actions, which can be performed by the system. These actions can either be observable actions, identified through unique labels, or unobservable actions, denoted by a special label τ. An extension of LTSs called IOLTSs introduces a distinction between input and output actions. It will be used to define semantics for action systems and to define the **ioco** conformance relation.

Definition 2.1 (Labelled Transition Systems).
A Labelled Transition System is a 4-tuple $\langle Q, q_0, \Sigma, \rightarrow \rangle$ where

- *Q is a countable, non-empty set,*
- *q_0 is the initial state,*
- *Σ is a countable set of labels and*
- *$\rightarrow \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ is the transition relation.*

A computation of a system consists of the application of the transition relation, repeated finitely often. That is, starting from the initial state q_0, a system repeatedly executes actions and by that changes its state. The transition relation encodes, which actions are possible in a given state and the post state reached by executing an action. Based on the transition relation, the generalised transition relation $\Longrightarrow$ can be defined, which generalises the transition relation to sequences of observable actions, also called traces. In the following, the short-hand notation $q \xrightarrow{l} q'$ will be used for $(q, l, q') \in \rightarrow$. For two sequences of observable actions $\sigma_1 \in \Sigma^*$ and $\sigma_2 \in \Sigma^*$, $\sigma_1 \cdot \sigma_2$ will denote the concatenation of these sequences and ϵ will denote an empty sequence.

Definition 2.2 (Generalised Transition Relation - LTS).
Let $p = \langle Q, q_0, \Sigma, \rightarrow \rangle$ be an LTS, and let $q, q', q'' \in Q$, $\lambda \in \Sigma$ and $\sigma \in \Sigma^*$. The generalised transition relation is the smallest set $\Longrightarrow \subseteq Q \times \Sigma^* \times Q$ satisfying the rules given below. As for the transition relation, a short-hand notation will be used, $(q, \sigma, q') \in \Longrightarrow$ will be abbreviated as $q \overset{\sigma}{\Rightarrow} q'$.

($\mathbf{T}\epsilon$) $q \overset{\epsilon}{\Rightarrow} q$

($\mathbf{T}\tau$) *if* $q \overset{\sigma}{\Rightarrow} q'$ *and* $q' \xrightarrow{\tau} q''$ *then* $q \overset{\sigma}{\Rightarrow} q''$

($\mathbf{T}\lambda$) *if* $q \overset{\sigma}{\Rightarrow} q'$ *and* $q' \xrightarrow{\lambda} q''$ *then* $q \overset{\sigma \cdot \lambda}{\Longrightarrow} q''$

In the following, further notations for LTSs shall be defined.

Definition 2.3 (Additional Notations - LTS).
Let $p = \langle Q, q_0, \Sigma, \rightarrow \rangle$ be an LTS and let $q \in Q$, $\mu \in \Sigma \cup \{\tau\}$ and $\sigma \in \Sigma^*$:

$$q \xrightarrow{\mu} =_{def} \exists q' \in Q : q \xrightarrow{\mu} q'$$
$$q \stackrel{\sigma}{\Rightarrow} =_{def} \exists q' \in Q : q \stackrel{\sigma}{\Rightarrow} q'$$
$$q \stackrel{\mu}{\not\rightarrow} =_{def} \neg\exists q' \in Q : q \xrightarrow{\mu} q'$$

As noted above, the definition of LTSs can be further refined to differentiate between input and output actions yielding the definition of IOLTSs and definitions based on IOLTSs as given in [45].

Definition 2.4 (Input Output Labelled Transition Systems).
An IOLTS is a 5-tuple $\langle Q, q_0, \Sigma_I, \Sigma_U, \rightarrow \rangle$ such that $\langle Q, q_0, \Sigma_I \cup \Sigma_U, \rightarrow \rangle$ is an LTS with $\Sigma_I \cap \Sigma_U = \emptyset$.

The labels in Σ_I represent the observable input actions and the labels in Σ_U represent the observable output actions, which are also referred to as observations. Analogously to LTSs, the special label τ with $\tau \notin \Sigma_I \cup \Sigma_U$ denotes an unobservable action, the set Q represents the states of a system and q_0 is the initial state of a system. Similarly, $q \xrightarrow{\mu} q'$ is used as an abbreviation for $(q, \mu, q) \in \rightarrow$.

Since **ioco** considers the absence of outputs to be an observation, it is necessary to introduce the notion of quiescence. A special quiescence label δ shall denote that it is impossible to produce outputs or to perform internal actions in a given state. A state $q \in Q$ is defined to be quiescent, denoted by $\delta(q)$, if $\forall o \in \Sigma_U \cup \{\tau\} : q \stackrel{o}{\not\rightarrow}$. The set including all observable actions and the quiescence observation shall be denoted by Σ_δ, which is given by $\Sigma_\delta = \Sigma_I \cup \Sigma_U \cup \{\delta\}$. Furthermore, the set of traces Σ_δ^* shall be referred to as the set of extended traces. Based on this, the suspension transition relation, which accounts for the observation of quiescence and extends the generalised transition relation, shall be defined:

Definition 2.5 (Suspension Transition Relation).
Let $p = \langle Q, q_0, \Sigma_I, \Sigma_U, \rightarrow \rangle$ be an IOLTS and let $q, q' \in Q$ and $\sigma \in \Sigma_\delta^*$. The suspension transition relation is the smallest relation $\Longrightarrow_\delta \subseteq Q \times \Sigma_\delta^* \times Q$ satisfying the rules $(\mathbf{T}\epsilon)$, $(\mathbf{T}\tau)$, $(\mathbf{T}\lambda)$, and $(\mathbf{T}\delta)$, where $\Longrightarrow$ is replaced by $\Longrightarrow_\delta$ in the first three rules. The new rule $(\mathbf{T}\delta)$ is given by:

$(\mathbf{T}\delta)$ *if $q \stackrel{\sigma}{\Rightarrow}_\delta q'$ and $\delta(q')$ then $q \stackrel{\sigma \cdot \delta}{\Longrightarrow}_\delta q'$*

Before it is possible to define the **ioco** conformance relation, further auxiliary functions need to be defined.

Definition 2.6 (Auxiliary functions for IOLTSs).
Let $p = \langle Q, q_0, \Sigma_I, \Sigma_U, \rightarrow \rangle$ an IOLTS and let $q \in Q$, $C \subseteq Q$ and $\sigma \in \Sigma_\delta^*$:

1. $Straces(q) =_{def} \left\{ \sigma \in \Sigma_\delta^* \middle| q \stackrel{\sigma}{\Rightarrow}_\delta \right\}$ *... the set of suspension traces*
2. $C \,\mathbf{after}\, \sigma =_{def} \bigcup_{q \in C} q \,\mathbf{after}\, \sigma$, *where* $q \,\mathbf{after}\, \sigma =_{def} \{q' | q \stackrel{\sigma}{\Rightarrow}_\delta q'\}$ *... the set of states reachable from C by executing σ*
3. $out(C) =_{def} \bigcup_{q \in C} out(q)$, *where* $out(q) =_{def} \left\{ \mu \middle| \mu \in \Sigma_U : q \xrightarrow{\mu} \right\} \cup \{\delta | \delta(q)\}$ *... the set of observations possible in states of C*
4. $der(q) =_{def} \left\{ q' \middle| \exists \eta \in \Sigma^* : q \stackrel{\eta}{\Rightarrow} q' \right\}$, *the set of states reachable by executing arbitrary traces of actions*

The **ioco** conformance relation is defined for implementations modelled as weakly input-enabled IOLTSs [45, 78][1]. An IOLTS $\langle Q, \Sigma_I, \Sigma_U, \rightarrow, q_0 \rangle$ is weakly input-enabled if and only if:

$$\forall q \in der(q_0) \forall \mu \in \Sigma_I : q \stackrel{\mu}{\Rightarrow}$$

[1] Tretmans actually uses Input Output Transition Systems(IOTSs) to model implementations, which are weakly input-enabled by definition.

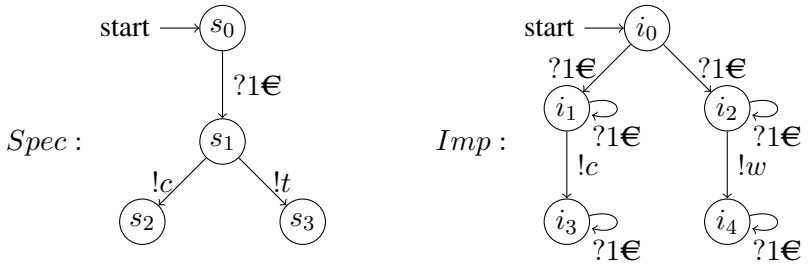


Figure 2.1: IOLTS-models of a coffee machine specification and an implementation.

Hence, from all reachable states it must either be possible to execute input actions or to reach states by executing unobservable actions in which it is possible to execute inputs. Stated differently, an implementation must not block any input, but rather accept all inputs.

Definition 2.7 (Input Output Conformance).
Let $\mathcal{S} = \langle Q_S, s_0, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS representing a specification, $\mathcal{P} = \langle Q_P, p_0, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ be a weakly input-enabled IOLTS representing an implementation and let $\mathcal{F} \subseteq Straces(s_0)$. $\mathcal{P}$ is $\mathbf{ioco}_\mathcal{F}$-conform to $\mathcal{S}$, denoted by $\mathcal{P}\ \mathbf{ioco}_\mathcal{F}\ \mathcal{S}$, iff

$$\forall \sigma \in \mathcal{F} : out(p_0\ \mathbf{after}\ \sigma) \subseteq out(s_0\ \mathbf{after}\ \sigma)$$

Furthermore, $\mathcal{P}\ \mathbf{ioco}\ \mathcal{S}$, iff

$$\forall \sigma \in Straces(s_0) : out(p_0\ \mathbf{after}\ \sigma) \subseteq out(s_0\ \mathbf{after}\ \sigma)$$

Hence, two versions of **ioco** are defined above [45, 78]. One is defined for all possible suspension traces of the specification and the other is defined with respect to a subset of those traces. Note that Tretmans and Frantzen et al. give slightly different definitions of $\mathbf{ioco}_\mathcal{F}$ [45, 78]. Although Tretmans' original definition does not require that $\mathcal{F} \subseteq Straces(s_0)$, this restriction is placed on $\mathcal{F}$ in Definition 2.7 because there is a corresponding restriction in the definition of **sioco** [45]. Informally speaking, an implementation must accept all inputs and it must not show observations, which are not allowed by the specification, after any of the considered traces.

Example 2.2 (Input Output Labelled Transition Systems and Input Output Conformance).
Figure 2.1 shows two IOLTS-models of coffee machines. They are similar to examples used by Weiglhofer et al. [82]. They define the input action ?1€ which denotes the insertion of one euro and they define output actions !c, !t and !w which denote the output of coffee, tea and water. Note that inputs are prefixed by question marks and outputs by exclamation marks.

Both models contain non-determinism: the model on the left may output coffee or tea upon receiving one euro, while the model on the right may choose to enter one of two states after receiving money. However, in those states the system chooses deterministically to either output coffee or water. The formal representation of the model on the left is an IOLTS $Spec = \langle Q_{Spec}, s_0, \Sigma_I, \Sigma_U, \rightarrow_{Spec} \rangle$ where $Q_{Spec} = \{s_0, s_1, s_2, s_3\}$,$\Sigma_I = \{?1€\}$, $\Sigma_U = \{!c, !t, !w\}$ and $\rightarrow_{Spec} = \{(s_0, ?1€, s_1), (s_1, !c, s_2), (s_1, !t, s_3)\}$. The model on the right corresponds to an IOLTS $Imp = \langle Q_{Imp}, i_0, \Sigma_I, \Sigma_U, \rightarrow_{Imp} \rangle$, where $Q_{Imp} = \{i_0, i_1, i_2, i_3, i_4\}$, $\Sigma_I = \{?1€\}$, $\Sigma_U = \{!c, !t, !w\}$ and $\rightarrow_{Imp} = \{(i_0, ?1€, i_1), \ldots\}$.

*Note that both IOLTSs define the same sets of actions and that Imp is input-enabled, that is, it accepts all inputs in all states. As a result, it is possible to interpret Imp as an implementation and to check whether it conforms to $Spec$ with respect **ioco**. Intuitively, Imp should not conform to $Spec$ since it may output water which is not allowed by $Spec$. This can be formalised as follows:*

$$Imp\ \mathbf{\sout{ioco}}\ Spec$$
$$\text{because } out(i_0\ \mathbf{after}\ ?1€) = \{!c, !w\} \not\subseteq out(s_0\ \mathbf{after}\ ?1€) = \{!c, !t\}$$
$$\text{and } ?1€ \in Straces(s_0)$$

The trace ?1€ leads to a situation where non-conforming behaviour can be observed.

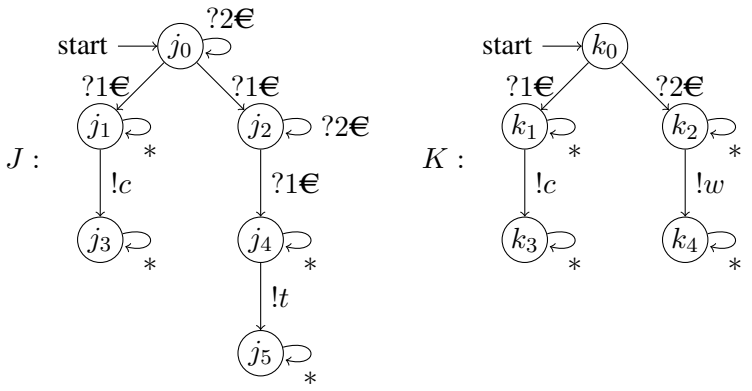


Figure 2.2: IOLTS-models of coffee machine implementations to demonstrate quiescence and implementation freedom for unspecified inputs.

Additionally to the example given above, two further models shall demonstrate situations which are particularly interesting in the context of **ioco**. The first shows that quiescence may cause conformance violations, while the second highlights that **ioco** allows for implementation freedom for unspecified inputs [78]. In other words, a conforming implementation may produce arbitrary outputs in response to inputs not foreseen by the specification because only traces of the specification are checked. This allows for the creation of partial specifications. These are specifications which only define requirements for explicitly modelled behaviour.

Example 2.3 (Non-conformance through Quiescence and Implementation Freedom).
Figure 2.2 shows two implementation models of coffee machines J and K. The set of inputs offered by these models contains ?2€ in addition to ?1€. In order to check conformance to the specification $Spec$ shown in Figure 2.1, ?2€ is also implicitly added to the inputs of $Spec$. To simplify representation, an edge labelled with $*$ denotes a set of edges, each labelled with one of the inputs.

It shall now be checked whether the implementation J shown on the left conforms to the specification. Consider the observations after the trace $?1€ \in Straces(s_0)$ for this purpose:

$$J \not\textbf{ioco} Spec$$

because $\delta(j_2)$

and therefore $out(j_0 \textbf{ after } ?1€) = \{!c, \delta\} \not\subseteq out(s_0 \textbf{ after } ?1€) = \{!c, !t\}$

Hence, J does not conform to $Spec$ because after receiving one euro it may wait for another euro and show quiescent behaviour.

Informally speaking, the implementation K shown on the right conforms to $Spec$ because it produces allowed outputs for all specified inputs. While it is sufficient to give one trace for showing non-conformance, all suspension traces need to be considered for showing conformance. These are given by $Straces(s_0) = \{\epsilon, \delta, \delta\delta, \ldots, ?1€, ?1€!c, ?1€!c\delta, ?1€!c\delta\delta, \ldots, ?1€!t, ?1€!t\delta, ?1€!t\delta\delta, \ldots\} \cup Q$. The set Q contains traces of the form $\delta\delta \ldots ?1€ \ldots$, that is, traces which start with quiescence followed by inserting one euro. These traces, however, will be ignored, because observing quiescence in the initial state does not change the behaviour of K. Repeated observation of quiescence will not be checked as well since the system state is not changed through quiescence. Showing conformance would not be possible otherwise because the set of suspension traces is of infinite size.

$$K \textbf{ ioco } Spec$$
$$\textit{because } out(k_0 \textbf{ after } \epsilon) = out(k_0 \textbf{ after } \delta) = \{\delta\} = out(s_0 \textbf{ after } \epsilon) = out(s_0 \textbf{ after } \delta)$$
$$\textit{and } out(k_0 \textbf{ after } ?1€) = \{!c\} \subseteq out(s_0 \textbf{ after } ?1€) = \{!c, !t\}$$
$$\textit{and } out(k_0 \textbf{ after } ?1€!c) = \{\delta\} = out(s_0 \textbf{ after } ?1€!c)$$
$$\textit{and } out(k_0 \textbf{ after } ?1€!c\delta) = \{\delta\} = out(s_0 \textbf{ after } ?1€!c\delta)$$
$$\textit{and } out(k_0 \textbf{ after } ?1€!t) = \{\} \subseteq out(s_0 \textbf{ after } ?1€!t) = \{\delta\}$$
$$\textit{and } out(k_0 \textbf{ after } ?1€!t\delta) = \{\} \subseteq out(s_0 \textbf{ after } ?1€!t\delta) = \{\delta\}$$
$$\textit{thus } \forall \sigma \in Straces(s_0) : out(k_0 \textbf{ after } \sigma) \subseteq out(s_0 \textbf{ after } \sigma)$$

Hence, producing the output $!w$ in response to the unspecified input ?2€ does not cause a conformance violation.

2.3 Action Systems

Action systems are introduced in the same way as for the USE-workshop [12].

Action systems were first defined by Back and Kurkio-Suonio as a modelling formalism for distributed systems [14]. The formalism was chosen as it can effectively be applied for modelling reactive systems [13] and because recently, it has also been adopted for model-based mutation testing [3, 4, 6, 7, 9].

Several variations of action systems exist, like object oriented action systems [23]. They also served as an inspiration for Event-B [1]. However, the action system formalism used in this thesis is more restricted than other variations. In some aspects it is similar to the Event-B language, but for instance does not support set-theoretic constructs to the extent as Event-B does.

Before a formal definition of action systems is given, their structure and execution shall be described informally [14]. Action systems basically define a system state, an initialisation of this state, and several guarded actions. The execution of an action system starts in the initial state which is manipulated by repeatedly executing actions. During this process one action is chosen at each step in a non-deterministic fashion from the set of enabled actions. An action is enabled if and only if its guard is satisfiable in the current state. The execution terminates when the set of enabled actions is empty.

2.3.1 Syntax

A definition of the concrete syntax of action systems is given in an adapted version of the Backus-Naur Form (BNF) in Figure 2.3. Overlines denote possibly empty repetitions of elements and bold-faced strings denote terminal symbols as in the syntax definition given by Aichernig and Jöbstl [6]. It abstracts away technical details like the separation of parameters by commas and it should be noted that the implementation also allows for the definition of further data types, but this is discussed in Section 6.1.

An action system definition starts with the definition of a name given as a capitalised identifier and contains the definition of types T, the declaration of state variables S, the initial state I and actions ACT.

The types block T consists of a list of type definitions, which associate type names with user-defined types. User-defined types can either be enumeration data types or range data types, with enumeration data types defining enumerations of constant symbols and range data types defining a closed interval of integers. These type definitions plus one additional predefined type $Bool$ are used in the state block S. The state variable and their types are defined in this block.

The *init*-block I contains one assignment with an expression over constant terms as right-hand side for each state variable. This block is followed by the actions block ACT which defines an arbitrary

$$
\begin{aligned}
AS &::= \texttt{def}\ capid\ \{T\ S\ I\ ACT\} & B &::= \{\overline{id\texttt{:=}\ E}\}\\
T &::= \texttt{types}\ \{\overline{capid\texttt{=}TYPE}\} & E &::= id \mid C \mid E\texttt{+}E \mid \texttt{!}E \mid \texttt{(}E\texttt{)}\\
TYPE &::= \texttt{[}int\texttt{..}int\texttt{]} \mid \texttt{[}\overline{capid\texttt{|}}capid\texttt{]} & &\quad \mid E\texttt{==}E \mid E\texttt{<}E \mid \ldots\\
S &::= \texttt{state}\ \{\overline{id\texttt{:}ty}\} & C &::= \texttt{True} \mid \texttt{False} \mid capid \mid int\\
I &::= \texttt{init}\ B & id &::= \text{identifier}\\
ACT &::= \texttt{actions}\ \{\overline{A}\} & capid &::= \text{capitalised identifier}\\
A &::= (\texttt{?} \mid \texttt{!} \mid \epsilon)\ id\ \texttt{(}\overline{id\texttt{:}ty}\texttt{)}\ \texttt{if}\ E\ \texttt{then}\ B & int &::= \text{integer number literal}
\end{aligned}
$$

Figure 2.3: The action system syntax

number of actions A. Every action definition consists of a label definition, a parameter list, a formula called guard and at most one assignment per state variable. The label definition optionally starts with a question or an exclamation mark and contains an identifier, which defines the name of the action. A question mark denotes the action as input, an exclamation mark as output and the absence of both denotes it as internal action.

Example 2.4 (A Definition of a Simple Action System - Adder).
The following listing shows an action system, which defines two input actions and one output action, but no internal action. It is a simple adder whose state consists of a single integer variable, which may be increased, overwritten, or displayed and reset. Despite being an artificial example it is well-suited to discuss concepts introduced in the following sections.

The type `SmallInt = [-128..127]` *is defined as the syntax definition given in Figure 2.3 mandates the definition of a range data type for integer variables. Furthermore, the actions define guards to illustrate their effects. The action* `add` *may for instance only be executed for non-negative* `p` *and if the sum of the state variable* `x` *and the parameter* `p` *is less than or equal to* `50`*.*

```
1  def Add
2  {
3    types
4    {
5      SmallInt = [-128..127];
6    }
7    state
8    {
9      x : SmallInt;
10   }
11   init
12   {
13     x := 0;
14   }
15   actions
16   {
17     ?add(p : SmallInt) if p >= 0 && x + p <= 50 then
18     {
19       x := x + p;
20     };
21     ?assignValue(p : SmallInt) if x + p >= 50 then
22     {
23       x := p;
24     };
25     !showAndReset(value: SmallInt) if !(x == 0) && value == x then
26     {
27       x := 0;
28     }
29   }
30 }
```

2.3.2 Semantics

As stated by Butler [29], the semantics of action systems is usually defined via weakest precondition formulas, but in the following, a semantics, which relates action systems to IOLTSs, will be given. The semantics and definitions will closely follow the style used for Input Output Symbolic Transition Systems(IOSTSs) [45]. This approach was taken because action systems can easily be translated to initialised IOSTSs and this way it is possible to use the work of Frantzen et al. [45] with some adaptations. In the following, an abstract syntax of action systems, which will be used subsequently, will be defined. Based on that, auxiliary functions and restrictions are introduced.

Definition 2.8 (Abstract Syntax of Action Systems).
An action system is a tuple $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$, where $\mathcal{V}$ is the set of state variables and $\mathcal{I}$ is the set of parameter variables[2]*, with $\mathcal{V} \cap \mathcal{I} = \emptyset$ and $Var = \mathcal{V} \cup \mathcal{I}$. $\Lambda = \Lambda_I \cup \Lambda_U$ is the set of action labels, with Λ_I being the set of input actions and Λ_U being the set of output actions. The constant $\tau \notin \Lambda$ denotes an internal action and $\Lambda_\tau = \Lambda \cup \{\tau\}$ contains all observable and internal actions. The initialisation of the action system is given by $\iota \in \mathfrak{T}(\emptyset)^{\mathcal{V}}$. The set $\rightarrow \subseteq \Lambda_\tau \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}}$ is the transition relation. For $(\lambda, \varphi, \rho) \in \rightarrow$, λ is called label, φ is called guard, ρ is the update mapping, which is defined by the assignments in the body of an action.*

The elements of the transition relation will either be referred to as actions or as transitions. Note that action labels may also be referred to as actions, if its possible distinguish labels and transitions in the given context. Observable action labels actually uniquely identify a single transition as noted below.

Similarly to [45], the following functions and vocabulary will also be used:

1. $\mathsf{arity} : \Lambda_\tau \rightarrow \mathbb{N}_0$ is the arity function, that is, it associates each action with its number of parameters.
2. The function para associates each action λ with a tuple of size $\mathsf{arity}(\lambda)$ containing the parameter variables for λ.
3. For all actions λ, para maps λ to a tuple of distinct parameter variables and for $(\lambda, \varphi, \rho) \in \rightarrow$ it holds that $\mathsf{free}(\varphi) \subseteq \mathcal{V} \cup \mathsf{para}(\lambda)$ and $\rho \in \mathfrak{T}(\mathcal{V} \cup \mathsf{para}(\lambda))^{\mathcal{V}}$. This means that the guard can only contain the parameters of the corresponding action and the state variables as free variables. The update mapping ρ is a mapping from state variables to terms over state variables and action parameters.

An action system must satisfy the following properties in order to be well-defined:

1. For internal actions τ, it must hold that $\mathsf{arity}(\tau) = 0$. This means internal actions must not have parameters. The same restriction is also placed on STSs in [45].
2. The transition relation $\rightarrow$ must contain exactly one element for each observable action label, that is, $\forall \lambda \in \Lambda \colon |\{(\lambda, \varphi, \rho) | (\lambda, \varphi, \rho) \in \rightarrow\}| = 1$ must hold.

Although it is required that action systems must not define multiple transitions with the same observable action label, non-determinism can be expressed through the use of internal actions labelled with τ.

Example 2.5 (Abstract Syntactical Representation of a Simple Action System - Adder).
Given the action system definition from Example 2.4, its abstract syntactical representation is $\mathcal{ADD}$, where $\mathcal{ADD} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ and

- $\mathcal{V} = \{x\}$, $\mathcal{I} = \{par\_add\_p, par\_assignValue\_p, value\}$,
- $\Lambda_I = \{?add, ?assignValue\}$, $\Lambda_U = \{!showAndReset\}$,

[2] Note that $\mathcal{I}$ is used rather than $\mathcal{P}$ to avoid confusion with power sets and because parameter variables correspond to interaction variables of Symbolic Transition Systems(STSs).

- $\iota = \{x \mapsto 0\}$,
- $\rightarrow = \{(?add, par\_add\_p \geq 0 \wedge x + par\_add\_p \leq 50, \{x \mapsto x + par\_add\_p\})\} \cup$
- $\{(?assignValue, x + par\_assignValue\_p \geq 50, \{x \mapsto par\_assignValue\_p\})\} \cup$
  $\{(!showAndReset, \neg(x = 0) \wedge value = x, \{x \mapsto 0\})\}$,
- $\mathsf{para}(?add) = (par\_add\_p)$,
- $\mathsf{para}(?assignValue) = (par\_assignValue\_p)$,
  $\mathsf{para}(!showAndReset) = (value)$, *and*
- $\mathsf{arity}(?add) = \mathsf{arity}(?assignValue) = \mathsf{arity}(!showAndReset) = 1$.

Note that by convention, the parameter variables are prefixed by par,the action name and underlines for separation, if the parameter names are not unique across actions. This is done because there needs to exists a tuple of distinct parameter variables for all actions. Since integers are used in this example, it is assumed that the underlying first-order structure is based on integers and supports the used functions and predicates. The restriction to the interval $[-200, 200]$*, which is defined using the concrete syntax, is added to the guards by the actual implementation. However, it is ignored for the example to keep it simple.*

A semantics of action systems based on an interpretation as IOLTSs is given below.

Definition 2.9 (Interpretation of Action Systems as IOLTSs).
Let $\mathcal{AS}$ *be an action system given by* $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$*. Its interpretation* $[\![\mathcal{AS}]\!]$ *as IOLTS is defined as* $[\![\mathcal{AS}]\!] = \langle Q, q_{init}, \Sigma_I, \Sigma_U, \rightarrow_{LTS} \rangle$*, where*

- $Q = \mathfrak{U}^{\mathcal{V}}$ *is the set of all states,*
- $q_{init} = \mathsf{eval} \circ \iota$ *is the initial state,*
- $\Sigma_I = \bigcup_{\lambda \in \Lambda_I} (\{\lambda\} \times \mathfrak{U}^{\mathsf{arity}(\lambda)})$ *is the set of input actions,*
- $\Sigma_U = \bigcup_{\lambda \in \Lambda_U} (\{\lambda\} \times \mathfrak{U}^{\mathsf{arity}(\lambda)})$ *is the set of output actions,*
- $\Sigma_\tau = \Sigma_I \cup \Sigma_U \cup \{\tau\}$ *is the set of all actions, and*
- $\rightarrow_{LTS} \subseteq \mathfrak{U}^{\mathcal{V}} \times \Sigma_\tau \times \mathfrak{U}^{\mathcal{V}}$ *is defined by the rule:*

$$\frac{(\lambda, \varphi, \rho) \in \rightarrow \qquad \varsigma \in \mathfrak{U}^{\mathsf{para}(\lambda)} \qquad \vartheta \cup \varsigma \models \varphi \qquad \vartheta' = (\vartheta \cup \varsigma)_{\mathsf{eval}} \circ \rho}{(\vartheta, (\lambda, \varsigma(\mathsf{para}(\lambda))), \vartheta') \in \rightarrow_{LTS}}$$

The fact that it is possible to define semantics for action systems based on LTSs can be used for the development of model-based testing tools and more specifically for **ioco** testing tools. Such an approach was for instance followed for the development of the explicit conformance checker Ulysses [3, 4] and also for the development of the **ioco** checking tool performed by the author of this thesis [58].

However, this definition will not be utilised directly for conformance checking in this thesis because it would not be possible to exploit the symbolic structure defined in action systems. It should illustrate the semantics of action systems by means of a well-known formalism rather than symbolically. Additionally, it allows to reason about action systems in terms of IOLTSs. Similar interpretations will also be given for other symbolically defined objects.

Example 2.6 (Interpretation of a Simple Action System - Adder).
This example addresses the interpretation of the action system given in Example 2.4 and 2.5 respectively. Let the action system $\mathcal{ADD}$ *be defined as in Example 2.5, its interpretation is given by*

$$[\![\mathcal{ADD}]\!] = \langle Q, q_{init}, \Sigma_I, \Sigma_U, \rightarrow_{LTS} \rangle, \text{ where}$$

$$q_{init} = \mathsf{eval} \circ \{x \mapsto 0\} = \{x \mapsto 0\}.$$

It should be noted that the constant 0 *to the left of the equality sign is an element of* $\mathfrak{T}(\emptyset)$ *(syntax) and the constant* 0 *at the right-hand side is an element of the universe* $\mathfrak{U}$ *(semantics). The sets* Q, Σ_I *and* Σ_U *are defined in a straight-forward way, but a transition in* $\rightarrow_{LTS}$ *shall be considered. The transition relation* $\rightarrow_{LTS}$ *contains for instance*

$$(\{x \mapsto 0\}, (?add, \{par\_add\_p \mapsto 10\}), \{x \mapsto 10\}).$$

It contains the given transition because

$$\{par\_add\_p \mapsto 10\} \in \mathfrak{U}^{\mathsf{para}(?add)},$$
$$\{x \mapsto 0\} \cup \{par\_add\_p \mapsto 10\} = \{x \mapsto 0, par\_add\_p \mapsto 10\},$$
$$\{x \mapsto 0, par\_add\_p \mapsto 10\} \models par\_add\_p \geq 0 \wedge x + par\_add\_p \leq 50 \text{ and}$$
$$\{x \mapsto 0, par\_add\_p \mapsto 10\}_{\mathsf{eval}} \circ \{x \mapsto x + par\_add\_p\} = \{x \mapsto 10\}.$$

Hence, the interpretation of $\mathcal{ADD}$ *contains a transition from the initial state to the state* $\{x \mapsto 10\}$ *labelled with the input action* $(?add, \{par\_add\_p \mapsto 10\})$.

2.4 Symbolic Execution

In this section, the technique of symbolic execution will be introduced. First, the basic idea behind it and its origins will be discussed. This is followed by concepts and terminology commonly used in the area of symbolic execution. The main part of this section will cover the fundamental principles of the symbolic execution of action systems.

2.4.1 Symbolic Execution on Implementation-Level

The technique of symbolically executing programs was first proposed in the nineteen-seventies. Early research in this area has, among others, been performed by Howden [53], Boyer et al. [26] and King [60]. It was developed to facilitate testing and debugging of software and can be seen as a compromise between formal proving and testing of programs. However, it took about three decades to be practically usable [32], because first generation tools suffered from efficiency problems. As a result, the interest in symbolic execution grew in the last few years. Two main reasons contributed to this development: (1) advances in the area of constraint solving and (2) the combination of concrete and symbolic execution. Successful tools, which pioneered the approach of combining concrete and symbolic execution are EXE [31] and DART [51]. Another prominent tool which applies such a combination is Pex [74]. However, this thesis will focus on purely symbolic execution.

The basic working principle of symbolic execution is to execute programs with symbolic values for inputs rather than with concrete values [60]. Inputs may for instance be values submitted by the user, or parameters of functions. It does not necessitate changes of programs or programming languages, but only changes to the execution semantics of programs. Importantly, the symbolic execution semantics of programs can be defined in a way such that it is a natural extension of the normal execution semantics. This means that, given a set of concrete input values K, the result of the symbolic execution of a program, in which all symbolic inputs are substituted by the values in K, is the same as the result of the execution using the values in K as inputs. For a program P, this property can formally be expressed via

$$\forall K \in \mathfrak{U}^X : K_{\mathsf{eval}}(E_s(P(X))) = E_c(P(K(X))), \text{ where}$$

the symbolic inputs are given by the tuple X, the concrete inputs are given by the valuation K, and the symbolic and concrete execution results are returned by the functions E_s and E_c. This is also illustrated by the commutative diagram in Figure 2.4.

In the following, important concepts will be described. The descriptions follow King's work [60] by for instance using a similar terminology. As mentioned above, symbolic execution uses symbolic values

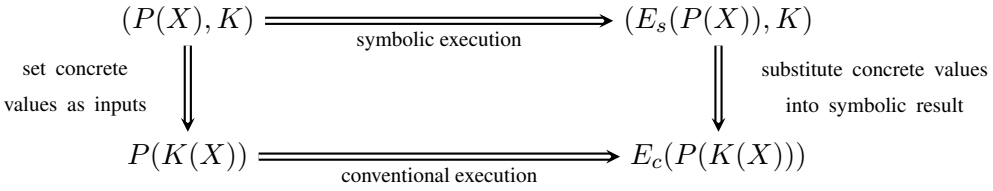


Figure 2.4: Commutativity of symbolic execution of a program P (adapted from King [60]).

for inputs. Since a symbolic value may represent any concrete value of the same type, expressions can not be evaluated as usual. Hence, expressions and more generally the execution state need to be treated in a symbolic way. The execution state consists of the statement counter, the values of program variables and a so-called path condition.

As in the normal execution semantics the statement counter points to the statement, which is currently executed. Differently from the execution with concrete values, symbolic values may be assigned to program variables. More generally, expressions involving program variables and symbolic inputs may be assigned to program variables. Since only inputs are treated symbolically, the current value of a program variable needs to be substituted when an expression is evaluated. Hence, the state of a program variable is an expression containing symbolic values denoting inputs. Analogously, the return value of a function is also an expression over symbolic values rather than a concrete value.

An example for the symbolic execution of a function is given in Figure 2.5. It is also used by King to illustrate symbolic execution [60]. The function calculates the sum of three numerical parameters A,B and C, which take symbolic values α_A, α_B and α_C respectively. In the figure, question marks denote that the initial values of program variables are unknown. The comments on the right show the symbolic state after executing the corresponding line if the line is an assignment. For the line containing the `return`-statement, the comment shows the symbolic return-value, that is, an expression over the symbolic inputs α_A, α_B and α_C.

```
1: function SUM(A,B,C)                    ▷ program variable state: X =?, Y =?, Z =?
2:     X ← A + B                          ▷ X = α_A + α_B, Y =?, Z =?
3:     Y ← B + C                          ▷ X = α_A + α_B, Y = α_B + α_C, Z =?
4:     Z ← X + Y − B                      ▷ X = α_A + α_B, Y = α_B + α_C, Z = α_A + α_B + α_C
5:     return Z                           ▷ Return α_A + α_B + α_C
6: end function
```

Figure 2.5: Symbolic execution of the Sum-function

Conditions of branching statements may reference symbolic values as well. In the general case, a condition may evaluate to true or false depending on the values taken by the symbolic inputs. Hence, both paths of the branching statement should be executed symbolically. A path condition records for each of the paths which constraints need to be satisfied in order for the path to be executed. As a result, the path condition is a conjunction of formulas over symbolic values. Considering the concrete example of an `IF`-statement with a condition c, the condition c is added to the path condition of the execution following the `THEN`-branch, while $\neg c$ is added for the `ELSE`-branch. Such branching statements where both paths are followed are called forking. Branching statements may also be non-forking if the path condition implies the condition or the negation of the condition. In these situations it is possible to uniquely determine which path needs to be followed. The normal execution for concrete values satisfying a given path condition follows the same path as the corresponding symbolic execution.

A (symbolic) execution tree may be created for a program, which represents the symbolic execution of a number of paths of the program. It contains nodes for each executed statement, which are branching if the corresponding statement is forking. Such a tree contains information about the current execution state for each node. In general, nodes are only created if the corresponding path conditions are satisfiable.

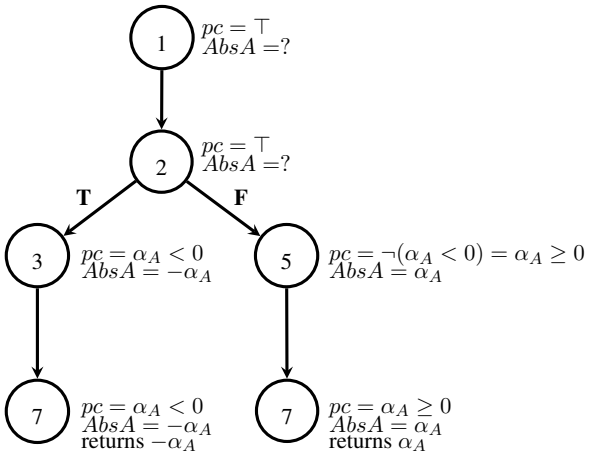


Figure 2.6: The symbolic execution tree for the Abs-function given by Algorithm 1. The statement numbers are displayed inside the nodes, the path condition and the variable values are displayed to the right of the nodes. A bold **T** (for true) denotes the arc for the `THEN`-branch of the `IF`-statement, while a bold **F** (for false) denotes the arc for the `ELSE`-branch.

However, a complete execution tree may still be of infinite size if the program contains loop statements. Consequently, the maximum number of executions of a loop or, more generally, the search depth, needs to be bounded. A symbolic execution tree for the Abs-function given by Algorithm 1 is depicted in Figure 2.6.

It remains to be discussed how symbolic execution could be utilised for testing. As suggested by Howden [53], a class of paths could be selected and symbolically executed for a given program. Afterwards concrete input values could be selected satisfying each of the path conditions corresponding to the executed paths. By that it is possible to generate test data which causes the previously selected paths to be executed. However, as implemented in the SELECT tool [26], the symbolic information may be used as well. A user could define assertions which need to hold in terms of symbolic values. The symbolic execution tool can check if there exist concrete values which violate those assertions via constraint solving. Hence, using this technique, testing would be performed at a symbolic level. This thesis is targeted towards this approach, thus execution and conformance checks of action systems are performed purely symbolically.

2.4.2 Symbolic Execution on Model-Level and Symbolic Input Output Conformance

Several concepts presented in the following have also been described for the USE-workshsop [12]. They include (indexed) parameter variables, symbolic states, the quiescence condition Δ and related concepts such as the equivalence condition for symbolic states.

Algorithm 1 The Abs-Function, which calculates and returns the absolute value of its parameter.

```
1: function ABS(A)
2:     if A < 0 then
3:         AbsA ← −A
4:     else
5:         AbsA ← A
6:     end if
7:     return AbsA
8: end function
```

Initially, symbolic execution was introduced to analyse and test programs. However, the symbolic execution of specifications was supported to a certain extent by first generation tools. As mentioned before, the SELECT tools was able to check symbolic assertions, which can be seen as specifications. This was supported by the EFFIGY tool [60] as well. Boyer et al. also identified the need for symbolic handling of specifications of sub routines to allow hierarchical testing by means of symbolic execution [26].

Since then, the symbolic execution technique has been adapted to support the execution of specifications given in a modelling language in a symbolic way. Since action systems are similar to IOSTSs, ideas and definitions developed for the symbolic execution of IOSTSs will be used and adapted for the symbolic execution of action systems. At first, the symbolic framework for model-based testing developed by Frantzen et al. [45] will be adapted for the symbolic execution of action systems. This will be followed by the definition of **sioco** conformance of action systems, which is also based on the definition given for IOSTSs in the symbolic framework for model-based testing. The section will be concluded by an adaption of the definition of state inclusion given by Gaston et al. [47].

Symbolic Execution of Action Systems

Essentially, the symbolic execution of action systems can be performed similarly to the symbolic execution of programs, thus concepts like the path condition may also be used on model-level. As in [45], symbolic traces will initially be used to describe the behaviour of action systems rather than symbolic execution trees. The general idea is to execute actions with symbolic values as parameters and to add the guards of executed actions to the path condition. Hence, the path condition is a formula over the initial state and symbolic parameters. A symbolic state vector will be included in the symbolic execution context for action systems as well. It forms the counterpart of the symbolic program variable values and is a mapping from state variables to terms over the initial state and symbolic parameters.

Unlike IOSTSs and programs, action systems do not define locations. Hence, it is not necessary to keep track of locations and the execution context consist solely of a path condition and a symbolic state vector. Note that similar to transitions, symbolic traces may be executed in an arbitrary state. As a result, the state variables $\mathcal{V}$ appear symbolically in both the path condition and symbolic state vector corresponding to a trace.

Since one action may occur multiple times on a trace, there is a need to distinguish the parameters used for different executions of a single action. Consequently, indexed sets of mutually disjoint parameter variables are introduced, where the index corresponds to the position of the action execution in the trace. The sets $\mathcal{I}_1, \mathcal{I}_2, \ldots$ are used to denote those parameter variables, $\widehat{\mathcal{I}}$ is defined by $\widehat{\mathcal{I}} = \bigcup_j \mathcal{I}_j$ and $\widehat{Var}$ is defined by $\widehat{Var} = \widehat{\mathcal{I}} \cup \mathcal{V}$. Additionally, a bijective variable-renaming $r_n \in \mathcal{I}_n^{\mathcal{I}}$ is assumed to exist. Based on this variable-renaming, an index-shifting function $s^{\gg i} \in \widehat{\mathcal{I}}^{\widehat{\mathcal{I}}}$ shall be defined for all $i \in \mathbb{N}$ by:

$$s^{\gg i}(x) = \begin{cases} (r_{j+i} \circ r_j^{-1})(x) & \ldots \exists j : x \in \mathcal{I}_j \\ x & \ldots \text{ else} \end{cases}$$

Furthermore, let $\widehat{\mathcal{I}}_i$ be the set of parameter variable with an index $\leq i$, that is, $\widehat{\mathcal{I}}_i = \bigcup_{k \leq i} \mathcal{I}_k$. The variable-renaming r_n assigns an index to parameters which denotes the position of the corresponding action in a trace, while the index-shifting function $s^{\gg i}$ adjust a previously set index. This may for instance be necessary if two traces are concatenated.

In the following, further concepts will be introduced. For this purpose, it will be assumed that an action system $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ is given.

The first concept to be discussed is the generalised transition relation which characterises symbolic traces of actions. More concretely, it captures the effects of executing a sequence of observable actions $\sigma \in \Lambda^*$. The executions of observable actions may be interleaved with executions of unobservable actions. As a result, the execution of a sequence may have multiple effects depending on the executed internal actions. This is expressed through the rule $\mathbf{S}\tau$.

Definition 2.10 (Generalised Transition Relation - Action Systems).
The generalised transition relation $\Rightarrow \subseteq \Lambda^* \times \mathfrak{F}(\widehat{Var}) \times \mathfrak{T}(\widehat{Var})^{\mathcal{V}}$ *is the smallest relation satisfying the following three rules:*

(Sϵ) $(\epsilon, \top, \mathsf{id}) \in \Rightarrow$

(Sτ) $(\sigma, \varphi \wedge \psi[\rho], [\rho] \circ \pi) \in \Rightarrow$ *if* $(\sigma, \varphi, \rho) \in \Rightarrow$ *and* $(\tau, \psi, \pi) \in \rightarrow$

(Sλ) $(\sigma \cdot \lambda, \varphi \wedge (\psi[r_n])[\rho], ([\rho] \circ ([r_n] \circ \pi))_{\mathcal{V}}) \in \Rightarrow$ *if* $(\sigma, \varphi, \rho) \in \Rightarrow$ *and* $(\lambda, \psi, \pi) \in \rightarrow$ *and* $n = length(\sigma) + 1$

The condition φ associated with an element (σ, φ, ρ) of the generalised transition relation will, following the terminology of the symbolic execution of programs, also be referred to as path condition.

Since implementations are considered to be weakly input-enabled in the context of **ioco** [78] and also in the context of **sioco** [45], another rule shall be introduced to circumvent this restriction. The reason is that this thesis deals with conformance checks between action systems for which input-enabledness cannot be assumed. As a result, the generalised transition relation for implementations is defined to adhere to the rule **Sa** given in Definition 2.11. This rule corresponds to the angelic completion described in [79], which is usually applied to make IOLTSs input-enabled. The angelic completion adds self-loops to states for all non-specified inputs. Stated differently, it ignores all non-specified inputs.

For action systems, this is achieved by allowing input actions to be executed in the way defined by Rule **S**λ or by Rule **Sa** given below. This rule states that an input action may be executed by ignoring the update mapping and by simultaneously negating its guard. Since weak input-enabledness allows internal actions to be performed before accepting inputs, a disjunction over all guards of τ-actions is formed, negated and added to the path condition via conjunction. As a result, an input action may not be executed with negated guard if internal actions are executable.

Definition 2.11 (Angelic Completion of Action Systems).
The generalised transition of implementations must also satisfy the rule:

(Sa) $(\sigma \cdot \lambda, \varphi \wedge \neg(\psi[r_{n+1}])[\rho] \wedge \neg\mu[\rho], \rho) \in \Rightarrow$ *if* $(\sigma, \varphi, \rho) \in \Rightarrow$ *and* $\exists \pi : (\lambda, \psi, \pi) \in \rightarrow$ *and* $\lambda \in \Lambda_I$ *and* $\mu = \bigvee_{(\tau,\eta,\gamma)\in\rightarrow} \eta$ *and* $n = length(\sigma)$

Example 2.7 (Generalised Transition Relation - Adder).
This example will again built upon Example 2.5 and illustrate the generalised transition relation by exemplarily listing elements of it. By definition, the generalised transition relation $\Rightarrow$ *contains*

$$(\epsilon, \top, \mathsf{id}) = (\epsilon, \top, \{x \mapsto x\})$$

and through an application of the rule **S**λ*, it contains*

$$\begin{aligned}(&\epsilon \cdot ?add, \top \wedge ((par\_add\_p \geq 0 \wedge x + par\_add\_p \leq 50)[r_1])[\mathsf{id}],\\ &([\mathsf{id}] \circ ([r_1] \circ \{x \mapsto x + par\_add\_p\}))_{\mathcal{V}})\end{aligned}$$

which can be simplified to

$$(?add, par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50, \{x \mapsto x + par\_add\_p_1\}).$$

Through another application of the rule **S**λ *with the* $?add$*-transition, it can be concluded that the generalised transition relation also contains*

$$\begin{aligned}(&?add \cdot ?add,\\ &par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50 \wedge\\ &((par\_add\_p \geq 0 \wedge x + par\_add\_p \leq 50)[r_2])[\{x \mapsto x + par\_add\_p_1\}],\\ &([\{x \mapsto x + par\_add\_p_1\}] \circ ([r_2] \circ \{x \mapsto x + par\_add\_p\}))_{\mathcal{V}})\end{aligned}$$

which is equal to

$$\begin{aligned}(?add?add,&\\ &par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50 \wedge\\ &par\_add\_p_2 \geq 0 \wedge x + par\_add\_p_1 + par\_add\_p_2 \leq 50,\\ &\{x \mapsto x + par\_add\_p_1 + par\_add\_p_2\}).\end{aligned}$$

If $\mathcal{ADD}$ would be considered to be an implementation, its generalised transition relation would satisfy the rule **Sa** *and thereby for instance contain*

$$\begin{aligned}&(\epsilon{\cdot}?add, \top \wedge \neg((par\_add\_p \geq 0 \wedge x + par\_add\_p \leq 50)[r_1])[\mathsf{id}], \mathsf{id})\\ &= (?add, \neg(par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50), \{x \mapsto x\})\end{aligned}$$

as well.

The concept of symbolic states shall now be introduced. These symbolic states correspond to sets of concrete states of an action system. As mentioned above, like the execution state in the symbolic execution of programs, they will consist of a path condition and a state vector in which the state of each state variable is described by a term over symbolic values.

Definition 2.12 (Symbolic States).
A symbolic state is a pair $(\varphi, \rho) \in \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}$. The pair element φ will be referred to as path condition and ρ will be referred to as symbolic state vector. An indexed symbolic state is a triple $(\varphi, \rho, i) \in \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0$, also written as $(\varphi, \rho)_i$, in which all indexed parameter variables occurring in the path condition φ and the symbolic state vector ρ have an index lower than or equal to i.

The term *path condition* may refer to the condition associated with an element of the generalised transition relation and to the condition associated with a symbolic state. However, it will usually be possible to infer from the context, which of the two concepts is considered. A symbolic state is said to be satisfiable, if its path condition is satisfiable. As indicated above, a symbolic state corresponds to a set of concrete states called interpretations of a symbolic state, which are defined below.

Definition 2.13 (Interpretation of Symbolic States).
Let $\eta = (\varphi, \rho)$ be a symbolic state. Its interpretation with respect to $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ is defined as $[\![\eta]\!]_\upsilon = \{\upsilon_{\mathsf{eval}} \circ \rho \mid \upsilon \models \varphi\}$. All possible interpretations are defined as $[\![\eta]\!] = \bigcup_{\upsilon' \in \mathfrak{U}^{\widehat{\mathcal{I}}}} [\![\eta]\!]_{\upsilon'}$.

Symbolic Input Output Conformance

Building upon the previous definitions, concepts necessary for the definition of **sioco** will now be introduced.

Since **ioco** considers quiescence as an observation of a system, quiescence should also be taken into account on the symbolic level. A state q of an IOLTS is quiescent if it is not possible to execute an output or an internal action in q [79]. Hence, the condition Δ for the observation of quiescence, with $\Delta \in \mathfrak{F}(\mathcal{V})$, can be given as by Frantzen et al. [45]:

$$\Delta =_{def} \bigwedge \{\neg \overline{\exists}_{\mathsf{para}(\lambda)} \psi \mid \exists \rho : (\lambda, \psi, \rho) \in \rightarrow \text{ with } \lambda \in \Lambda_U \cup \{\tau\}\}$$

The observation of quiescence will be treated similarly to ordinary actions and denoted by the label δ, thus $(\delta, \Delta, \mathsf{id})$ will be used as quiescence transition, for which $\mathsf{arity}(\delta) = 0$ is assumed. Since the absence of observations does not update the state, the identity function is used as update mapping. A symbolic state (φ, ρ) is thus quiescent for the valuations $\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}}$, for which $\varsigma \models \varphi \wedge \Delta[\rho]$.

Example 2.8 (Quiescence Observation - Adder).
For the action system $\mathcal{ADD}$ given in Example 2.5, the condition for observing quiescence is given by

$$\begin{aligned}\Delta &= \neg\overline{\exists}_{\mathsf{para}(!showAndReset)}(\neg(x=0)\wedge value = x)\\ &\Leftrightarrow \neg\exists value : x\neq 0 \wedge value = x \Leftrightarrow x = 0.\end{aligned}$$

The quiescence condition is only satisfiable for $x = 0$, thus quiescence may for instance be observed in the initial state.

Next, $\mathbf{out}_\mathrm{s}$ the symbolic counterpart of the out-function used for **ioco** as defined in Section 2.2 will be given. The $\mathbf{out}_\mathrm{s}$-function calculates a set of symbolic observations for a given symbolic state. A symbolic observation may either be an output action or the quiescence observation. Similar to symbolic states, symbolic observations also correspond to sets of concrete observations, interpretations of the observations. A symbolic observation consists of three parts, an action label, the path condition of the state in which the observation is made and the guard of the observation. The path condition needs to be considered as well as the guard, because the execution history may also constrain the variables mentioned in the guard. More formally, a symbolic observation is a triple $(\lambda_\delta, \varphi, \psi) \in \mathcal{O}$, where $\mathcal{O}$, the set of all symbolic observations, is defined as $(\Lambda_U \cup \{\delta\}) \times \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{F}(\widehat{\mathcal{I}} \cup \mathcal{I})$, with $\mathsf{free}(\psi) \subseteq \mathsf{para}(\lambda_\delta) \cup \widehat{\mathcal{I}}$.

Definition 2.14 (Symbolic Observations for a Symbolic State).
Let (φ, ρ) be a symbolic state. The function $\mathbf{out}_\mathrm{s}$ is defined as:

$$\mathbf{out}_\mathrm{s}((\varphi,\rho)) =_{def} \{(\lambda, \varphi, \psi[\rho]) \mid \exists\pi : (\lambda,\psi,\pi) \in \rightarrow \wedge\ \lambda \in \Lambda_U\} \cup \{(\delta, \varphi, \Delta[\rho])\}$$

For sets C of symbolic states, $\mathbf{out}_\mathrm{s}$ shall be defined as:

$$\mathbf{out}_\mathrm{s}(C) =_{def} \bigcup_{(\varphi,\rho)\in C} \mathbf{out}_\mathrm{s}((\varphi,\rho))$$

Now it is also possible to define the symbolic suspension transition relation, which is the smallest relation $\Rightarrow_\delta \subseteq \Lambda_\delta^* \times \mathfrak{F}(\widehat{Var}) \times \mathfrak{T}(\widehat{Var})^{\mathcal{V}}$ satisfying the rules $\mathbf{S}\epsilon$, $\mathbf{S}\tau$,$\mathbf{S}\lambda$ and $\mathbf{S}\delta$, where $\Lambda_\delta = \Lambda_I \cup \Lambda_U \cup \{\delta\}$ and $\mathbf{S}\delta$ is defined below. As for the generalised transition relation, the symbolic suspension transition relation of implementations must also satisfy $\mathbf{Sa}$.

Definition 2.15 (Rule $\mathbf{S}\delta$).
The rule $\mathbf{S}\delta$ is given by:

$\mathbf{S}\delta\ \ (\sigma\cdot\delta, \varphi\wedge\Delta[\rho], \rho) \in \Rightarrow_\delta$ *if* $(\sigma,\varphi,\rho) \in \Rightarrow_\delta$

If quiescence is observed during the symbolic execution of action systems, the δ-label is added to the trace of actions and Δ is added to the path condition. The rule $\mathbf{S}\delta$ defines a condition for determining if a trace of actions leads to a state, which may show quiescent behaviour. The guard Δ restricts the set of valuations satisfying the path condition.

Based on the symbolic suspension transition relation, the notion of symbolical extended traces will now be introduced. First, $\mathsf{var} : \Lambda_\delta^* \rightarrow \mathcal{P}(\widehat{\mathcal{I}})$ is defined for sequences. Given a sequence of actions $\sigma \in \Lambda_\delta^*$, it computes the sets of parameter variables, which may be referenced by σ. It is defined as:

$$\mathsf{var}(\sigma) = \begin{cases}\emptyset & \text{if } \sigma = \epsilon\\ \mathsf{var}(\sigma') & \text{if } \sigma = \sigma'\cdot\delta\\ \mathsf{var}(\sigma') \cup \{r_{length(\sigma)}(\nu) \mid \nu \in \mathsf{para}(\lambda)\} & \text{if } \sigma = \sigma'\cdot\lambda\end{cases}$$

The set of symbolic extended traces is given by $\mathcal{E} = \{(\sigma,\varphi) \in \Lambda_\delta^* \times \mathfrak{F}(\widehat{Var}) \mid \mathsf{free}(\varphi) \subseteq \mathcal{V} \cup \mathsf{var}(\sigma)\}$. Since this thesis focuses on testing for **sioco** between action systems, which are initialised, and does not

aim to provide a general framework like Frantzen et al. [45], another set $\mathcal{E}_i$ shall be introduced. The set of initialised symbolic extended traces $\mathcal{E}_i$ is defined as $\{(\sigma, \varphi) \in \Lambda_\delta^* \times \mathfrak{F}(\widehat{\mathcal{I}}) \mid \mathsf{free}(\varphi) \subseteq \mathsf{var}(\sigma)\}$ and shall contain all symbolic extended traces from the initial state. That is, the state variables freely occurring in φ for $(\sigma, \varphi) \in \mathcal{E}$ are initialised based on the initialisation ι. Hence, it is possible to derive $\mathcal{E}_i$ from $\mathcal{E}$ through $\mathcal{E}_i = \{(\sigma, \varphi[\iota]) \mid (\sigma, \varphi) \in \mathcal{E}\}$. Like symbolic states, which correspond to sets of concrete states, initialised symbolic extended traces also correspond to sets of concrete traces. Their interpretation as such is given below.

Definition 2.16 (Interpretation of Symbolic Extended Traces).
Let $(\sigma, \varphi) \in \mathcal{E}_i$ be an initialised symbolic extended trace and let $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ be a valuation of parameter variables. The interpretation of (σ, φ) with respect to υ is given by:

$$[\![(\sigma, \varphi)]\!]_\upsilon =_{def} \{\mathbf{etrace}_\upsilon(\sigma) \mid \upsilon \models \varphi\}$$

where $\mathbf{etrace}_\upsilon$ is defined as:

$$\mathbf{etrace}_\upsilon(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \mathbf{etrace}_\upsilon(\sigma') \cdot \delta & \text{if } \sigma = \sigma' \cdot \delta \\ \mathbf{etrace}_\upsilon(\sigma') \cdot (\lambda, \upsilon(r_{length(\sigma)}(\mathsf{para}(\lambda)))) & \text{if } \sigma = \sigma' \cdot \lambda \end{cases}$$

The set of all interpretations $[\![(\sigma, \varphi)]\!]$ is defined by $[\![(\sigma, \varphi)]\!] =_{def} \bigcup_{\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}}} [\![(\sigma, \varphi)]\!]_\upsilon$. For sets E with $E \subseteq \mathcal{E}_i$, the set of all interpretations shall be defined as $[\![E]\!] = \bigcup_{(\sigma,\varphi) \in E} [\![(\sigma, \varphi)]\!]$.

Based on the definition of initialised symbolic extended traces, the set of initialised symbolic suspension traces of an action system shall be defined as $Straces_s = \{(\sigma, \varphi[\iota]) \in \mathcal{E}_i \mid \exists \rho : (\sigma, \varphi, \rho) \in \Rightarrow_\delta\}$. For the set of all interpretations of $Straces_s$ it holds that $[\![Straces_s]\!] = Straces(\mathsf{eval} \circ \iota)$. This means that interpretations of initialised symbolic suspension traces directly correspond to suspension traces executable from the initial state of the IOLTS-interpretation of an action system.

Before the **sioco** conformance relation can be defined for action systems, a symbolic $\mathbf{after}$-function needs to be defined. As before, a less general definition than given by Frantzen et al. [45] will be used. It is, however, well-suited to **sioco** conformance checking. The function $\mathbf{after}_{\mathrm{s_{init}}}$ associates initialised symbolic extended traces with indexed symbolic states. In other words, it returns all symbolic states reachable by a given traces. It corresponds to the $\mathbf{after}_{\mathrm{s}}$-function [45] evaluated for the symbolic state $(l_{init}, \top, \iota)_0$, where l_{init} is the initial location of an IOSTS.

Definition 2.17 ($\mathbf{after}_{\mathrm{s_{init}}}$-function).
Let $(\sigma, \chi) \in \mathcal{E}_i$ be an initialised symbolic extended trace, the function $\mathbf{after}_{\mathrm{s_{init}}} : \mathcal{E}_i \to \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0$ is defined as follows:

$$\mathbf{after}_{\mathrm{s_{init}}}(\sigma, \chi) =_{def} \left\{(\chi \wedge \psi[\iota], [\iota] \circ \pi)_{length(\sigma)} \;\middle|\; (\sigma, \psi, \pi) \in \Rightarrow_\delta\right\}$$

Note that by convention, the path conditions of symbolic states and elements of the generalised transition relation are usually referred to via the greek letter φ, while χ usually denotes the condition corresponding to an initialised symbolic extended trace. This distinction in naming reflects the different nature of both conditions: χ may be chosen arbitrary, while φ depends on the considered action system. Nevertheless, letters other than φ may also be used for path conditons if necessary.

Example 2.9 (After-function - Adder).
This example shall demonstrate the application of the $\mathbf{after}_{\mathrm{s_{init}}}$-function by determining the symbolic states reachable by executing the initialised symbolic extended traces $(?add, \top)$ and $(?add?add, \top)$ using the action system $\mathcal{ADD}$ defined in Example 2.5. The action system shall therefore be interpreted as a specification, that is, rule $\mathbf{Sa}$ is ignored.

$$
\begin{aligned}
\mathbf{after}_{\mathrm{s_{init}}}(?add, \top) &= \{(\top \wedge \psi[\iota], [\iota] \circ \pi)_{length(\sigma)} \mid (\sigma, \psi, \pi) \in \Rightarrow_\delta\} \\
&= \{((par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50)[\iota], \\
&\quad [\iota] \circ \{x \mapsto x + par\_add\_p_1\})_1\} \ldots \textit{ see also Example 2.7} \\
&= \{((par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50)[\{x \mapsto 0\}], \\
&\quad [\{x \mapsto 0\}] \circ \{x \mapsto x + par\_add\_p_1\})_1\} \\
&= \{(par\_add\_p_1 \geq 0 \wedge 0 + par\_add\_p_1 \leq 50, \\
&\quad \{x \mapsto 0 + par\_add\_p_1\})_1\} \\
&= \{(par\_add\_p_1 \geq 0 \wedge par\_add\_p_1 \leq 50, \{x \mapsto par\_add\_p_1\})_1\} \\
\mathbf{after}_{\mathrm{s_{init}}}(?add?add, \top) &= \{((par\_add\_p_1 \geq 0 \wedge x + par\_add\_p_1 \leq 50 \wedge par\_add\_p_2 \geq 0 \wedge \\
&\quad x + par\_add\_p_1 + par\_add\_p_2 \leq 50)[\iota], \\
&\quad [\iota] \circ \{x \mapsto x + par\_add\_p_1 + par\_add\_p_2\})_2\} \\
&= \{(par\_add\_p_1 \geq 0 \wedge par\_add\_p_1 \leq 50 \wedge par\_add\_p_2 \geq 0 \wedge \\
&\quad par\_add\_p_1 + par\_add\_p_2 \leq 50, \\
&\quad \{x \mapsto par\_add\_p_1 + par\_add\_p_2\})_2\}
\end{aligned}
$$

Since $\mathcal{ADD}$ is deterministic and considered to be a specification, the cardinality of $\mathbf{after}_{\mathrm{s_{init}}}(\sigma, \chi)$ is exactly one. As a result, exactly one state $(\varphi, \rho)_1$ is reached by executing $?add$ and exactly one state $(\varphi, \rho)_2$ is reached by executing $?add?add$.

Now, the **sioco** conformance relation can be defined as follows:

Definition 2.18 (Symbolic Input Output Conformance).
Let $\mathcal{F}_s$ be a set of initialised symbolic extended traces for an action system $\mathcal{AS}_S$ representing a specification, with $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$ and $[\![\mathcal{F}_s]\!] \subseteq Straces(\mathsf{eval} \circ \iota_S)$. An implementation given as an action system $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$, for which rule $\mathbf{Sa}$ given in Definition 2.11 is fulfilled, with $\mathcal{V}_S \cap \mathcal{V}_P = \emptyset$, is $\mathbf{sioco}_{\mathcal{F}_s}$-conform to $\mathcal{AS}_S$ (written $\mathcal{AS}_P$ $\mathbf{sioco}_{\mathcal{F}_s}$ $\mathcal{AS}_S$), iff

$$
\begin{aligned}
&\forall (\sigma, \chi) \in \mathcal{F}_s \; \forall \lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\forall}_{\widehat{\mathcal{I}} \cup \mathcal{I}} \left(\Phi_P(\lambda, \sigma) \wedge \chi \rightarrow \Phi_S(\lambda, \sigma)\right) \\
&\qquad \textit{where } \Phi_{AS}(\lambda, \sigma) = \bigvee \{\varphi \wedge \psi \mid (\lambda, \varphi, \psi) \in \mathbf{out}_{\mathrm{s}}(\mathbf{after}_{\mathrm{s_{init}}}(\sigma, \top))\} \\
&\qquad\quad \textit{and } \Phi_{AS} \textit{ denotes that the calculation of } \Phi_{AS} \textit{ is based on } \rightarrow_{AS}
\end{aligned}
$$

Hence, an implementation is $\mathbf{sioco}_{\mathcal{F}_s}$-conform to a specification if and only if all conditions for observations of the implementation imply the conditions derived for the specification. This corresponds to the requirement that the set of observations of the implementation must be a subset of the observations of the specification in the non-symbolic definition of **ioco**. Like the original definition of **ioco**, **sioco** is defined with respect to a set of traces of the specification. The restriction $[\![\mathcal{F}_s]\!] \subseteq Straces(\mathsf{eval} \circ \iota_S)$ requires that the traces, for which **sioco** conformance is checked, are executable by the specification.

Equivalence of Symbolic States

Gaston et al. gave a definition of state inclusion [47], which will be adapted to define equivalence between two symbolic states. Their definition is based on all possible interpretations of symbolic states, thus the definition of symbolic state equivalence shall be based on the set of all interpretations as well. Hence, two symbolic states $\eta = (\varphi, \rho)$ and $\eta' = (\varphi', \rho')$ are defined to be equivalent, denoted by $\eta \equiv \eta'$, if $[\![\eta]\!] = [\![\eta']\!]$. A model-theoretic definition of equivalence is given in the next definition.

Definition 2.19 (Equivalence of Symbolic States).
Let $\eta = (\varphi, \rho)_i$ and $\eta' = (\varphi', \rho')_j$ be two symbolic indexed states. $\eta \equiv \eta'$ iff:

- *if for all $\zeta \in \mathfrak{U}^{\mathcal{V}}$ and a $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ such that $\zeta \cup \upsilon \models (\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi)$ there exists a $\upsilon' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ such that $\zeta \cup \upsilon' \models (\bigwedge_{x \in \mathcal{V}} x = \rho'(x) \wedge \varphi')$*
- *and if for all $\zeta' \in \mathfrak{U}^{\mathcal{V}}$ and a $\upsilon'' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ such that $\zeta' \cup \upsilon'' \models (\bigwedge_{x \in \mathcal{V}} x = \rho'(x) \wedge \varphi')$ there exists a $\upsilon''' \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ such that $\zeta' \cup \upsilon''' \models (\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi)$*

Thus in order to determine if two symbolic states η and η' are equivalent, it is necessary to check if both $[\![\eta]\!] \subseteq [\![\eta']\!]$ and $[\![\eta']\!] \subseteq [\![\eta]\!]$ hold. The first condition checks if $[\![\eta]\!] \subseteq [\![\eta']\!]$ while the second checks if $[\![\eta']\!] \subseteq [\![\eta]\!]$.

The reason for adapting state inclusion rather applying it directly shall now be discussed. It is necessary to consider the intended application areas of state inclusion/equivalence checks for this purpose. Gaston et al. define a criterion for reducing the search space during test cases generation based on state inclusion [47]. The rationale behind this approach is that for two symbolic states η and η', with $[\![\eta]\!] \subseteq [\![\eta']\!]$, every behaviour possible in η is also possible in η'. As a result, the symbolic execution tree may be pruned at η. This thesis proposes a similar approach but with a slightly different purpose and based on symbolic state equivalence.

Section 4.1 discusses an optimisation which suggests the precomputation of a symbolic execution tree for the specification. The tree should encode information about all executable action sequences of bounded length. In this context, symbolic state equivalence checks are basically used to detect loops in the symbolic execution tree. As a result, it is possible to explore the precomputed symbolic execution tree and whenever a "pruned" state is detected, a set of reachable post-states in the tree can be determined because an exactly equivalent state has been explored. Hence, the symbolic execution tree is transformed into a general directed graph. If state inclusion would be checked instead, it would merely be possible to determine a set of states which may be reachable. However, this raises the question as to why it is necessary to determine post-states of "pruned" states. The reason is: while a specification will show known behaviour during further execution, an implementation may show unknown, possibly non-conforming behaviour.

In conclusion, it is necessary to apply state equivalence checks rather than inclusion checks because the intended application areas are different. While Gaston et al. aim at reducing search space [47], this thesis proposes to precompute a symbolic execution graph with loops in order to explore it afterwards.

Example 2.10 (Equivalence of Symbolic States - Adder).
Given the action system $\mathcal{ADD}$ as defined in Example 2.5, it shall be determined if the symbolic state $(\varphi, \rho)_1$, reached by executing $?add$, is equivalent to $(\varphi', \rho')_2$, which is reached by executing $?add?add$. Their structure is shown in Example 2.9.

By investigating the symbolic states, it can be observed that in state $(\varphi', \rho')_2$, the variable x is set to $par\_add\_p_1 + par\_add\_p_2$ which must fulfil $0 \leq par\_add\_p_1 + par\_add\_p_2 \leq 50$. In state $(\varphi', \rho')_1$, the variable x is set to $par\_add\_p_1$, which must fulfil $0 \leq par\_add\_p_1 \leq 50$. Since $par\_add\_p_2$ can be set to zero, the variable x can take the same set of concrete values in both states. The symbolic states correspond to the set of concrete states $\{\{x \mapsto 0\}, \{x \mapsto 1\}, \ldots, \{x \mapsto 50\}\}$. Since their interpretations are equal, the symbolic states $(\varphi, \rho)_1$ and $(\varphi', \rho')_2$ are equivalent.

This subsection introduced symbolic execution semantics for action systems. The fact that it is possible to define concrete interpretations of symbolic objects highlights that symbolic execution is indeed an generalisation of normal execution. Finally, definitions of the **sioco** conformance relation and symbolic state equivalence have been given.

3 Symbolic Input Output Conformance Checking

This chapter will focus on checking of **sioco** conformance. After a definition of bounded **sioco**, theoretical concepts will be introduced, including a formal definition of the product graph, which forms the basis of the **sioco** checking algorithm. In the second section of this chapter, algorithmic aspects of the **sioco** checking approach followed in this thesis will be presented. More concretely, a basic version of the **sioco** checking algorithm will be defined and discussed. Optimisations built upon this algorithm will be presented in Chapter 4. In order to be generally applicable, all algorithms will be defined using pseudo code in a technology-independent way.

In Section 2.4.2, **sioco** modulo a set $\mathcal{F}_s$ was defined. However, one goal of this thesis is to implement an **sioco** checker, which given a depth d, a specification action system and an implementation action system,

- either finds a trace of length $\leq d$ leading to a state in which non-conformance may be observed
- or performs a complete verification of conformance up to depth d.

In other words, bounded model-checking for **sioco** conformance should be performed [20]. Consequently, the definition of **sioco** will be adapted yielding $\mathbf{sioco}_d$, in the following also referred to as bounded **sioco**. This version of **sioco** considers all bounded sequences of observable actions executable by the specification. Although $\mathbf{sioco}_d$ is mainly used as conformance relation in this thesis, the term **sioco** will rather be used instead in subsequent sections, if the bound on the maximum depth is not relevant. In order to able to formulate a condition for $\mathbf{sioco}_d$, an auxiliary definition of $Straces(s)_d$ is needed. The set $Straces(s)_d$ denotes the set of suspension traces of length l from state s in an IOLTS, where $l \leq d$. It is defined as follows:

$$Straces(s)_d =_{def} \{\sigma | \sigma \in Straces(s) \wedge length(\sigma) \leq d\}$$

Since suspension traces hide internal actions, only observable actions are taken into account for the bound on the trace length. In other words, an arbitrary number of internal actions may be executed by a specification and an implementation considered for an $\mathbf{sioco}_d$-conformance check.

Definition 3.1 (Bounded Symbolic Input Output Conformance).
Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$ be an action system and let $d \in \mathbb{N}_0$ be a bound. An implementation given as an action system $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$, for which rule **Sa** *given in Definition 2.11 is fulfilled, with $\mathcal{V}_S \cap \mathcal{V}_P = \emptyset$, is $\mathbf{sioco}_d$-conform to $\mathcal{AS}_S$ (written $\mathcal{AS}_P$ $\mathbf{sioco}_d$ $\mathcal{AS}_S$), iff:*

$$\begin{aligned} &\mathcal{AS}_P \ \mathbf{sioco}_{\mathcal{F}_s} \ \mathcal{AS}_S \\ &\quad \textit{where } \mathcal{F}_s \textit{ is a set of initialised symbolic suspension traces} \\ &\quad \textit{such that } [\![\mathcal{F}_s]\!] = Straces(\mathsf{eval} \circ \iota_S)_d \end{aligned}$$

While **sioco** is originally defined with respect to a set of initialised symbolic extended traces, $\mathbf{sioco}_d$ considers explicitly only initialised symbolic suspension traces because of the restriction placed on $[\![\mathcal{F}_s]\!]$. Since the goal is to derive distinguishing test cases, a test for non-conformance rather than for conformance must be implemented. Consequently, the condition defined above needs to be negated for the implementation of the **sioco** checker, yielding a non-conformance condition.

Definition 3.2 (Non-conformance Condition for Bounded sioco).
Let $\mathcal{AS}_S, d$ and $\mathcal{AS}_P$ be defined as in Definition 3.1 and let Φ_{AS} be defined as in Definition 2.18, $\mathcal{AS}_P$ is not $\mathbf{sioco}_d$-conform to $\mathcal{AS}_S$, iff

$$\begin{aligned} \exists (\sigma, \chi) \in \mathcal{F}_s \ \exists \lambda \in \Lambda_U \cup \{\delta\} : \ & \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} \left(\Phi_P(\lambda, \sigma) \wedge \chi \wedge \neg \Phi_S(\lambda, \sigma) \right) \\ & \textit{where } \mathcal{F}_s \textit{ is a set of initialised symbolic suspension traces} \\ & \textit{such that } [\![\mathcal{F}_s]\!] = Straces(\mathsf{eval} \circ \iota_S)_d \end{aligned}$$

Let $c = \exists\lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} (\Phi_P(\lambda, \sigma) \wedge \chi \wedge \neg\Phi_S(\lambda, \sigma))$ be the non-conformance condition without existential quantification over traces. Essentially, to achieve the goal of either finding a trace to non-conformance or performing a verification of conformance, the condition c needs to be checked for all initialised symbolic suspension traces of bounded length. If non-conformance is detected for some trace, the trace can be analysed to determine why non-conformance exists or it can be used for test case generation. However, by showing that c is not satisfiable for any of the traces and thereby showing that the non-conformance condition is unsatisfiable, it can be concluded that the conformance condition is a tautology. Hence, this approach is able to verify conformance up to a given bound although it is a testing approach.

3.1 Symbolic Execution and Conformance Testing Concepts

Concepts presented in this section form the main part of the paper submitted to the USE-workshop [12]. An unbounded version of the symbolic execution tree defines the symbolic execution semantics of action systems and most importantly, the deterministic product graph and related theory is discussed as well.

In this section, further concepts based on the symbolic execution of action systems shall be introduced formally. These concepts shall help to bridge the gap between the formal definition of the **sioco** conformance relation and the **sioco** checking algorithm. The first two concepts are symbolic execution trees of bounded depth and symbolic state equivalence classes. Following these, two types of symbolic product graphs will be introduced. A simple product graph, based on the suspension transition relation defined in Section 2.4.2 and on the non-conformance condition given in Definition 3.2, shall present the general idea behind the usage of product graphs for conformance checking. The second type of product graph is introduced along with concepts relevant to it. This version of product graph shall provide a more practical non-conformance condition and serve as the basis for the implementation of an **sioco** checking algorithm. A definition of unsafe states concludes this section.

3.1.1 Symbolic Execution Tree

The concept of symbolic execution trees of action systems is inspired by execution trees created for programs [60]. A symbolic execution tree shall, starting from an initial state, encode the effects of symbolically executing arbitrary actions. Hence a symbolic execution tree may, alternatively to the symbolic suspension transition relation, be used to define symbolic execution semantics of action systems. It is actually closer to the original implementation as it highlights how symbolic states are changed through the execution of actions. However, for practical reasons, a depth bound shall be introduced. This limits the applicability of information contained in the symbolic execution tree to the execution of action sequences of finite length.

Definition 3.3 (Bounded Symbolic Execution Tree of an Action System).
Let $k > 0$ be an upper bound on the depth of the symbolic execution tree, $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow_{AS} \rangle$ be an action system, $Q \subseteq \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0$ be a set of indexed symbolic states with an index $\leq k$ and let $T \subseteq Q \times (\Lambda_\tau \cup \{\delta\}) \times Q$ be the set of edges of the symbolic execution tree. For $((\varphi, \rho, i), \lambda, (\varphi', \rho', j)) \in T$, the abbreviation $(\varphi, \rho)_i \xrightarrow{\lambda} (\varphi', \rho')_j$ will be used. The sets Q and T are defined to be the smallest sets satisfying the following rules:

Initial state:

$$\frac{}{(\top, \iota)_0 \in Q}$$

Execution of actions:

$$\frac{(\varphi,\rho)_n \in Q \qquad n < k \qquad (\lambda,\psi,\pi) \in \rightarrow_{AS} \qquad \lambda \neq \tau \qquad \varphi' = \varphi \wedge (\psi[r_{n+1}])[\rho] \qquad \rho' = ([\rho] \circ ([r_{n+1}] \circ \pi))\nu \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{n+1}} : \varsigma \models \varphi'}{(\varphi',\rho')_{n+1} \in Q \qquad (\varphi,\rho)_n \xrightarrow{\lambda} (\varphi',\rho')_{n+1}}$$

Observation of quiescence:

$$\frac{(\varphi,\rho)_n \in Q \qquad n < k \qquad \varphi' = \varphi \wedge \Delta[\rho] \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_n} : \varsigma \models \varphi'}{(\varphi',\rho')_{n+1} \in Q \qquad (\varphi,\rho)_n \xrightarrow{\delta} (\varphi',\rho)_{n+1}}$$

Execution of internal actions:

$$\frac{(\varphi,\rho)_n \in Q \qquad n \leq k \qquad (\tau,\psi,\pi) \in \rightarrow_{AS} \qquad \varphi' = \varphi \wedge \psi[\rho] \qquad \rho' = [\rho] \circ \pi \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_n} : \varsigma \models \varphi'}{(\varphi',\rho')_n \in Q \qquad (\varphi,\rho)_n \xrightarrow{\tau} (\varphi',\rho')_n}$$

Note that the indexes of states directly correspond to the execution depth, at which they have been detected. By convention, states reached by executing internal actions are considered to be at the same depth level as their pre-states, so the index value is not increased for such states. This is allowed by the definition of indexed symbolic states, as internal actions do not have parameters and thereby do not introduce new indexed parameter variables. Although this is also true for the quiescence observation, the symbolic state index is increased as it is observable.

Symbolic execution trees may not have finite size, if a symbolic state may be reached in which it is possible to execute an infinite sequence of internal actions. Following the definition of strong convergence of LTS given by Tretmans [79], action systems for which it is impossible to reach such a state shall be called convergent and otherwise divergent.

Example 3.1 (Symbolic Execution Tree).
Let $\mathcal{ADD}$ be an action system defined as in Example 2.5. The symbolic execution tree bounded by $k = 2$ created for $\mathcal{ADD}$ contains $q_0 = (\top, \{x \mapsto 0\})$ as initial state. The edge $(q_0, ?add, q_1)$ connects the initial state to one of its successors

$$q_1 = (par\_add\_p_1 \geq 0 \wedge par\_add\_p_1 \leq 50, \{x \mapsto par\_add\_p_1\}).$$

Furthermore, it contains another edge $(q_1, ?add, q_2)$, with

$$\begin{aligned} q_2 = (&par\_add\_p_1 \geq 0 \wedge par\_add\_p_1 \leq 50 \wedge par\_add\_p_2 \geq 0 \wedge \\ &par\_add\_p_1 + par\_add\_p_2 \leq 50, \\ &\{x \mapsto par\_add\_p_1 + par\_add\_p_2\}). \end{aligned}$$

This part of the symbolic execution tree is depicted in Figure 3.1. The complete tree contains transitions for other actions and quiescence as well.

Based on symbolic execution trees, the notion of symbolic state equivalence classes shall be introduced, which will be used for the optimisation of the **sioco** checker implementation.

Definition 3.4 (Symbolic State Equivalence Classes).
Let $Q \subseteq \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0$ be a set of indexed symbolic states, define a set EQ as the quotient set $EQ = Q/\equiv$, containing symbolic state equivalence classes, with the equivalence relation $\equiv$ given by Definition 2.19.

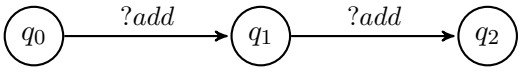


Figure 3.1: The part of the symbolic execution tree of the $\mathcal{ADD}$ action system described in Example 3.1.

3.1.2 Introduction to Product Graphs

While the non-conformance condition for **sioco** is derived from the conformance condition given by Frantzen et al. [45], the implementation for checking **sioco** conformance follows a similar approach as Weiglhofer and Wotawa [83], who define a product graph for checking **ioco** conformance. They explore the defined product graph "on the fly" and return a diagnostic sequence of actions leading to a state, where non-conformance may be observed, if the checked IOLTSs are not **ioco**-conform. Analogously, the check for **sioco** non-conformance shall be done by implicitly exploring a symbolic product graph. However, differently from the definition of the product graph for checking **ioco**, the symbolic product graph shall not contain explicit transitions to a $pass$-state. These would either be added for inputs accepted by the implementation but not by the specification, or for observations allowed by the specification, but not by the implementation. Beside transitions to system states, the symbolic product graph shall only contain transitions to $fail$-states. Like in Weiglhofer and Wotawa's definition [83], such $fail$-states shall denote that non-conforming behaviour may be observed in the pre-state. Each of the $fail$-states consists of the $fail$-label and the condition, which must be satisfied in order to observe non-conformance.

In the following definition, the notion of product graphs will be introduced. Although this version of product graph is not actually used in the implementation of the **sioco** conformance check, it shall present the idea behind product graphs in a concise way. Furthermore, it represents a connection between the symbolic trace semantics discussed in Section 2.4.2 and the second version of product graph, which forms the basis of the implementation.

Roughly speaking, the product graph is the product of the symbolic execution trees of a specification and an implementation, where internal actions are hidden and conformance is checked at each node. If non-conformance is detected, a transition to a $fail$-state is added.

Definition 3.5 (Product graph).
Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$ be an action system representing a specification, let $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$ be an action system representing an implementation, thus its symbolic suspension transition relations adheres to rule $\mathbf{Sa}$ *given in Definition 2.11, with $\mathcal{V}_S \cap \mathcal{V}_P = \emptyset$. Let $\Rightarrow_{\delta_S}$ and $\Rightarrow_{\delta_P}$ be their respective symbolic suspension transition relations and let $Q_S \subseteq \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S}$ and $Q_P \subseteq \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_P}$ be sets of symbolic states reachable from the the initial state by executing $\mathcal{AS}_S$ and $\mathcal{AS}_P$ respectively. The symbolic synchronous product graph $\mathcal{AS}_P \times_{sioco} \mathcal{AS}_S$ is a tuple $SP = \langle Q, q_{init} \rightarrow_{SP}, \Lambda_I, \Lambda_U \rangle$ where $Q \subseteq Q_P \times Q_S \times \Lambda_\delta^*$, $\Lambda_\delta = \Lambda_I \cup \Lambda_U \cup \{\delta\}$, $\rightarrow_{SP} \subseteq (Q_P \times Q_S) \times \Lambda_\delta \times ((Q_P \times Q_S) \cup (\{fail\} \times \mathfrak{F}(\widehat{\mathcal{I}} \cup \mathcal{I})))$ and $q_{init} = ((\top, \iota_P), (\top, \iota_S), \epsilon)$. For $(q, \lambda, q') \in \rightarrow_{SP}$, the abbreviation $q \xrightarrow{\lambda}_{SP} q'$ will be used. The transition relation $\rightarrow_{SP}$ and the set Q are defined as the smallest sets, satisfying the following rules:*

Initial state:

$$\frac{}{q_{init} \in Q}$$

Execution of actions:

$$\frac{\begin{array}{c} q \in Q \qquad q = ((\varphi_P, \rho_P), (\varphi_S, \rho_S), \sigma) \\ (\lambda, \phi_P, \pi_P) \in \rightarrow_{\delta_P} \qquad (\lambda, \phi_S, \pi_S) \in \rightarrow_{\delta_S} \qquad length(\lambda) = 1 \qquad n = length(\sigma) \\ \varphi'_P = \varphi_P \wedge (\phi_P[s^{\gg n}])[\rho_P] \qquad \varphi'_S = \varphi_S \wedge (\phi_S[s^{\gg n}])[\rho_S] \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} \colon \varsigma \models \varphi'_P \wedge \varphi'_S \\ \rho'_P = ([\rho_P] \circ ([s^{\gg n}] \circ \pi_P))\nu \qquad \rho'_S = ([\rho_S] \circ ([s^{\gg n}] \circ \pi_S))\nu \end{array}}{((\varphi'_P, \rho'_P), (\varphi'_S, \rho'_S), \sigma \cdot \lambda) \in Q \qquad q \xrightarrow{\lambda}_{SP} ((\varphi'_P, \rho'_P), (\varphi'_S, \rho'_S), \sigma \cdot \lambda)}$$

Detection of non-conformance:

$$\frac{\begin{array}{c} q \in Q \\ q = ((\varphi_P, \rho_P), (\varphi_S, \rho_S), \sigma) \qquad \lambda \in \Lambda_U \cup \{\delta\} \qquad \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} (\Phi_P(\lambda, \sigma) \wedge \varphi_S \wedge \neg \Phi_S(\lambda, \sigma)) \end{array}}{q \xrightarrow{\lambda}_{SP} (fail, \Phi_P(\lambda, \sigma) \wedge \varphi_S \wedge \neg \Phi_S(\lambda, \sigma))}$$

The action system $\mathcal{AS}_P$ is not **sioco**-conform to $\mathcal{AS}_S$, iff there exists a path from the initial state q_{init} to a $fail$-state. This path together with the condition associated with the $fail$-state serves as a witness of non-conformance. This term is actually borrowed from the area of model checking [20]. It should be noted that the condition $\exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi'_P \wedge \varphi'_S$ in the second rule may be rewritten to $\exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi'_S$ because $\textbf{sioco}_d$ is defined for all bounded symbolic suspension traces of the specification. Thus, only the path condition φ'_S of the specification must be satisfiable. However, checking if the path condition of the implementation is satisfiable as well, may be seen as a first optimisation. This optimisation still results in a complete **sioco** check because if some trace is not executable for the implementation, the non-conformance condition will not be satisfiable for this trace. Symbolic implementation states (φ'_P, ρ'_P) for which there exists no symbolic specification state (φ'_S, ρ'_S) such that $\varphi'_P \wedge \varphi'_S$ is satisfiable will also be discussed in Section 4.9 by introducing the notion of irrelevant states.

Note that non-conformance is checked for the path condition φ_S of a specification state instead for a condition associated with an initialised symbolic suspension trace. This is a valid condition because it corresponds to a trace executable by the specification.

Although this version of product graph is well-suited for giving another condition for **sioco** non-conformance and for defining witnesses of non-conformance, it hides a great amount of complexity. This complexity is contained in the definition of Φ_{AS} and in the symbolic suspension transition relation. Hence, another version of the product graph will be given below.

3.1.3 Deterministic Product Graph

The version of product graph presented in the following will be referred to as deterministic product graph and makes steps involved in the implementation of the **sioco** conformance check more explicit. These include the exploration of the product graph only up to a given depth and the explicit calculation of τ-closures. However, prior to giving the definition of the deterministic product graph, some auxiliary functions will be defined.

τ-closure

First, a symbolic τ-closure function shall be defined. It works by calculating a set of symbolic states reachable by executing internal actions.

Definition 3.6 (τ-closure).
Let $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ be an action system and $(\varphi, \rho) \in \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}$ be a symbolic state. The τ-closure of (φ, ρ) is defined as $\tau_{cl}(\{(\varphi, \rho)\})$, where:

$$\tau_{cl} : \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \rightarrow \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}})$$
$$\tau_{cl} : S \mapsto S \cup \tau_{cl}(\tau_{reach}(S))$$
$$\textit{where } \tau_{reach} : \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \rightarrow \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}})$$
$$\tau_{reach} : S \mapsto \bigcup_{(\varphi,\rho) \in S} \left\{ (\varphi \wedge \psi[\rho], [\rho] \circ \pi) \;\middle|\; (\tau, \psi, \pi) \in \rightarrow, \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \psi[\rho] \right\}$$

This definition of the τ-closure will obviously only terminate if τ_{reach} returns the empty set after a finite number of recursive applications of τ_{cl}. It is not applicable for models, which contain τ-loops. In the context of action systems, τ-loops may be defined as follows: an action system contains a τ-loop, if there exists a symbolic state from which it is possible to reach an equivalent state by executing a sequence of internal actions. In general, specifications may be assumed to not contain loops consisting of internal actions as is assumed by Tretmans [78]. Nevertheless, it is possible to define τ_{cl} in a way such that it can be applied regardless of the existence of τ-loops.

$$\tau_{cl} : S \mapsto S \cup \tau_{cl}(\{s \mid s \in \tau_{reach}(S) \wedge \neg \exists s' \in pred(s) \cap S : s \equiv s' \wedge s'\})$$
where $pred(s) =$ set of predecessors of s

Using this definition, a state s will not be explored further if an equivalent state s' is already in the closure and a predecessor of s. It must be a predecessor, because only predecessors place the same or looser restrictions on parameters of observable actions executed before. If this constraint would not be used, concrete states S equivalent to concrete states S' might implicitly be ignored, although being reached by different traces than S'[1].

It should be noted though that considering the interpretation of action systems, a restriction requiring that action systems must not contain τ-loops is not equivalent to the restriction placed on LTSs by Tretmans [78], which requires that all compositions of internal transitions must be finite. The following example highlights the difference between both restrictions.

Example 3.2 (τ-loops in Action System Interpretation).
Given is an action system $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ using integers as data, where:

- $\mathcal{V} = \{x\}$,
- $\mathcal{I} = \{i\}$,
- $\Lambda_I = \{?input\}$, $\Lambda_U = \{\}$,
- $\iota = \{x \mapsto 2\}$ *and*
- $\rightarrow = \{(?input, i = 0 \vee i = 1, \{x \mapsto i\}), (\tau, x \neq 2, \{x \mapsto x + x\})\}$.

Consider the interpretations of the symbolic states reached by the execution of action $?input$ in the initial state $(\top, \{x \mapsto 2\})$ followed by the execution of the internal action. The execution of $?input$ leads to the state

$$\eta = (i_1 = 0 \vee i_1 = 1, \{x \mapsto i_1\})$$

and the execution of the internal action leads to

$$\eta' = ((i_1 = 0 \vee i_1 = 1) \wedge i_1 \neq 2, \{x \mapsto i_1 + i_1\}).$$

The interpretations are given by

$$[\![\eta]\!] = \{\{x \mapsto 0\}, \{x \mapsto 1\}\} \text{ and } [\![\eta']\!] = \{\{x \mapsto 0\}, \{x \mapsto 2\}\},$$

thus $[\![\mathcal{AS}]\!]$ contains the transition $(\{x \mapsto 0\}, \tau, \{x \mapsto 0\})$, which represents a τ-loop. Since $\eta \not\equiv \eta'$, the execution of the given sequence of actions does not show a τ-loop in $\mathcal{AS}$ as defined for action systems. Hence, the interpretation of an action system $\mathcal{AS}$ may contain τ-loops although $\mathcal{AS}$ does not contain any.

In the definition of the concrete syntax given in Figure 2.3, data types are restricted to be finite. As a result, it is guaranteed that there is only a finite number of symbolic state equivalence classes, where equivalence is defined as in Definition 2.19. It follows that the τ-closure algorithm terminates after at most n steps, where n is the number of symbolic state equivalence classes.

Compound Symbolic States

Since determinisation will be performed explicitly in the following, operations are performed on sets of symbolic states, rather than on single states. Such sets of symbolic states will also be referred to as *compound symbolic states*, for which interpretations in terms of concrete states and an equivalence condition will be given.

[1] The definition of τ_{cl} given for the USE-workshop [12] does not contain the predecessor constraint and may lead to mutants being wrongfully detected as conforming.

Definition 3.7 (Compound Symbolic States).
A compound symbolic state is a non-empty set of symbolic states, thus it is an element of the set $\mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \setminus \{\emptyset\}$. *In the following, compound symbolic states will implicitly be assumed to be non-empty, thus the empty set will not be excluded explicitly. An indexed compound symbolic state is a set of indexed symbolic states, where all states in the set share the same index. A state* $s \in \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0)$, *such that* $\forall(\varphi, \rho, j) \in s : j = i$ *for some* $i \in \mathbb{N}_0$ *may be written as* s_i.

Since compound symbolic states are sets of symbolic states, the τ-closure function may be seen as a function mapping compound symbolic states to compound symbolic states. It should be noted that the τ-closure function may be applied on indexed and on non-indexed compound symbolic states in the same way because the execution of internal actions does not introduce new parameter variables. As a result, for an indexed compound symbolic state κ_i, the compound state calculated by $\tau_{cl}(\kappa_i)$ can be defined to have index i as well.

Analogously to symbolic states, the equivalence condition is based on all possible interpretations of compound symbolic states, which are defined below.

Definition 3.8 (Interpretations of Compound Symbolic States).
Let κ *be a compound symbolic state, its interpretation with respect to* $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ *is defined as* $[\![\kappa]\!]_\upsilon = \bigcup_{(\varphi,\rho)\in\kappa}\{\upsilon_{\mathsf{eval}} \circ \rho \mid \upsilon \models \varphi\}$. *All possible interpretations are given by* $[\![\kappa]\!] = \bigcup_{\upsilon\in\mathfrak{U}^{\widehat{\mathcal{I}}}}[\![\kappa]\!]_\upsilon$, *or equivalently by* $[\![\kappa]\!] = \bigcup_{\eta\in\kappa}[\![\eta]\!]$.

Let κ and κ' be two compound symbolic states, κ is equivalent to κ', denoted as $\kappa \equiv_{com} \kappa'$, if $[\![\kappa]\!] = [\![\kappa']\!]$. Equivalence between compound symbolic states may also be denoted as $\kappa \equiv \kappa'$ rather than as $\kappa \equiv_{com} \kappa'$, if it is possible to tell from the context that κ and κ' are compound symbolic states. A similar approach as for symbolic states shall be followed to define an equivalence condition for compound symbolic states, with only one difference: the union of interpretations of all symbolic states in a compound state will be encoded as a disjunction. This disjunction ranges over formulas formed from the symbolic states.

Definition 3.9 (Equivalence of Compound Symbolic States).
Let κ_i *and* κ_j *be two indexed compound symbolic states.* $\kappa_i \equiv \kappa_j$ *iff:*

- *if for all* $\zeta \in \mathfrak{U}^{\mathcal{V}}$ *and a* $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ *such that* $\zeta \cup \upsilon \models \bigvee_{(\varphi,\rho)_i\in\kappa_i}(\bigwedge_{x\in\mathcal{V}} x = \rho(x) \wedge \varphi)$ *there exists a* $\upsilon' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ *such that* $\zeta \cup \upsilon' \models \bigvee_{(\varphi,\rho)_j\in\kappa_j}(\bigwedge_{x\in\mathcal{V}} x = \rho(x) \wedge \varphi)$
- *and if for all* $\zeta' \in \mathfrak{U}^{\mathcal{V}}$ *and a* $\upsilon'' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ *such that* $\zeta' \cup \upsilon'' \models \bigvee_{(\varphi,\rho)_j\in\kappa_j}(\bigwedge_{x\in\mathcal{V}} x = \rho(x) \wedge \varphi)$ *there exists a* $\upsilon''' \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ *such that* $\zeta' \cup \upsilon''' \models \bigvee_{(\varphi,\rho)_i\in\kappa_i}(\bigwedge_{x\in\mathcal{V}} x = \rho(x) \wedge \varphi)$

The disjunction $\bigvee_{(\varphi,\rho)\in\kappa}\varphi$ over the path conditions of all states in a compound symbolic state κ is also called path condition of κ.

Product States

A deterministic product graph contains *product states* which are pairs of compound symbolic states. As for the other two types of states, interpretations of product states as well as an equivalence condition will be defined. The equivalence condition will be used in the actual implementation.

Definition 3.10 (Product States).
A product state is a pair of two compound symbolic states with disjoint sets of state variables, thus it is an element of the set $\mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_\mathcal{P}}) \times \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_\mathcal{S}})$, *where* $\mathcal{V}_P \cap \mathcal{V}_S = \emptyset$. *An indexed product state is a pair of two indexed compound symbolic states with the same index. An indexed product state* (κ_i, μ_i) *may also be written as* $(\kappa, \mu)_i$.

Since a product state is intended to be a pair consisting of an implementation and a specification compound symbolic state, the definition requires that the state variables of the two sub-states must be disjoint. In the following, the convention will be used, that the left pair element is a state of an action system modelling the implementation, while the right pair element is a state of the specification action system. For a product state (κ, μ), the path condition is the conjunction of the path conditions of the contained compound symbolic states κ and μ, thus it is given by

$$pc((\kappa,\mu)) = \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P \right) \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S \right).$$

A product state is said to be satisfiable, if its path condition is satisfiable. The definition of product state interpretations shall now be given as follows:

Definition 3.11 (Interpretations of Product States).
Let (κ, μ) be a product state, its interpretation with respect to a valuation υ is defined as $[\![(\kappa,\mu)]\!]_\upsilon = \bigcup_{(\varphi,\rho)\in\kappa}\{\upsilon_{\mathsf{eval}} \circ \rho \mid \upsilon \models \varphi\} \times \bigcup_{(\psi,\pi)\in\mu}\{\upsilon_{\mathsf{eval}} \circ \pi \mid \upsilon \models \psi\}$. The set of all interpretations is given by $[\![(\kappa,\mu)]\!] = \bigcup_{\upsilon\in\mathfrak{U}^{\widehat{\mathcal{I}}}}[\![(\kappa,\mu)]\!]_\upsilon$.

In the following the equivalence condition for two product state (κ, μ) and (κ', μ') will be given. It will be denoted as $(\kappa,\mu) \equiv_{prod} (\kappa',\mu')$ or as $(\kappa,\mu) \equiv (\kappa',\mu')$ if it is possible to tell from the context that both operands are product states. Analogously to the other types of states: $(\kappa,\mu) \equiv_{prod} (\kappa',\mu')$ if $[\![(\kappa,\mu)]\!] = [\![(\kappa',\mu')]\!]$. Therefore, the equivalent alternative definition of the equivalence condition for product states will also be based on interpretations. It needs to be considered though that a product state contains two sets of symbolic state vectors. As a result, pairs of valuations of state variables are needed.

Definition 3.12 (Equivalence of Product States).
Let (κ_i, μ_i) and (κ_j, μ_j) be two indexed product states. $(\kappa_i,\mu_i) \equiv (\kappa_j,\mu_j)$ iff:

- *if for all $(\zeta,\xi) \in \mathfrak{U}^{\mathcal{V}_P} \times \mathfrak{U}^{\mathcal{V}_S}$ and a $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ such that*
  $\zeta \cup \upsilon \models \bigvee_{(\varphi,\rho)_i\in\kappa_i}(\bigwedge_{x\in\mathcal{V}_P} x = \rho(x) \wedge \varphi)$ and $\xi \cup \upsilon \models \bigvee_{(\varphi,\rho)_i\in\mu_i}(\bigwedge_{x\in\mathcal{V}_S} x = \rho(x) \wedge \varphi)$
  there exists a $\upsilon' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ such that
  $\zeta \cup \upsilon' \models \bigvee_{(\varphi,\rho)_j\in\kappa_j}(\bigwedge_{x\in\mathcal{V}_P} x = \rho(x) \wedge \varphi)$ and $\xi \cup \upsilon' \models \bigvee_{(\varphi,\rho)_j\in\mu_j}(\bigwedge_{x\in\mathcal{V}_S} x = \rho(x) \wedge \varphi)$
- *and if for all $(\zeta',\xi') \in \mathfrak{U}^{\mathcal{V}_P} \times \mathfrak{U}^{\mathcal{V}_S}$ and a $\upsilon'' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ such that*
  $\zeta' \cup \upsilon'' \models \bigvee_{(\varphi,\rho)_j\in\kappa_j}(\bigwedge_{x\in\mathcal{V}_P} x = \rho(x) \wedge \varphi)$ and $\xi' \cup \upsilon'' \models \bigvee_{(\varphi,\rho)_j\in\mu_j}(\bigwedge_{x\in\mathcal{V}_S} x = \rho(x) \wedge \varphi)$
  there exists a $\upsilon''' \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ such that
  $\zeta' \cup \upsilon''' \models \bigvee_{(\varphi,\rho)_i\in\kappa_i}(\bigwedge_{x\in\mathcal{V}_P} x = \rho(x) \wedge \varphi)$ and $\xi' \cup \upsilon''' \models \bigvee_{(\varphi,\rho)_i\in\mu_i}(\bigwedge_{x\in\mathcal{V}_S} x = \rho(x) \wedge \varphi)$

The path condition of a product state has been defined as the conjunction of the path conditions of the compound symbolic states which compose the product state. Hence, it is required to hold for all product states in the product graph. The reason for this requirement shall be formalised in the following proposition which will be shown to hold at the end of this subsection.

Proposition 3.1 (Relevance of Product States).
Let (κ, μ) be a product state. If its path condition is unsatisfiable, that is, it holds that

$$\left(\bigvee_{(\gamma,\pi)\in\kappa} \gamma \right) \wedge \left(\bigvee_{(\gamma,\pi)\in\mu} \gamma \right) \leftrightarrow \bot,$$

then (κ, μ) is not relevant for the non-conformance check and thus may be ignored.

Definition of Deterministic Product Graph

Before another definition of the product graph will be given, two further auxiliary functions need to be defined, which describe how indexed compound symbolic states are changed through the execution of visible actions. The first function concerns the actual execution of an action, while the second concerns the "negated" execution. The latter actually ignores inputs for performing an *angelic completion* of the implementation, as described for LTSs by Tretmans [79]. This is done in order to make implementations input-enabled.

Since the angelic completion adds self-loops for undefined inputs to states of LTSs, the negated execution $exec_{neg}$ must not perform a state update. As implementations in the context of **sioco** are considered to be weakly input-enabled, $exec_{neg}$ should take into account that it is not necessary to add self-loops for inputs i if a state may be reached by executing internal actions, in which i is enabled. For this reason, the function expects the disjunction over the guards of all internal actions as third parameter which denotes a condition for executing an internal action. Furthermore, both functions take a compound symbolic state as first parameter and the guard of an action as second parameter. The $exec$-function takes the state update mapping of an action as third parameter.

$$
\begin{aligned}
exec &: \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}} \to \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \\
exec &: (\kappa_i, \psi, \pi) \mapsto \{(\varphi', \rho')_{i+1} \mid (\varphi, \rho)_i \in \kappa_i \wedge \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{n+1}} : \varsigma \models \varphi'\} \\
&\quad \text{where } \varphi' = \varphi \wedge (\psi[r_{i+1}])[\rho] \text{ and } \rho' = ([\rho] \circ ([r_{i+1}] \circ \pi))_{\mathcal{V}} \\
exec_{neg} &: \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \times \mathfrak{F}(Var) \times \mathfrak{F}(Var) \to \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \\
exec_{neg} &: (\kappa_i, \psi, \zeta) \mapsto \{(\varphi', \rho)_{i+1} \mid (\varphi, \rho)_i \in \kappa_i \wedge \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{n+1}} : \varsigma \models \varphi'\} \\
&\quad \text{where } \varphi' = \varphi \wedge \neg(\psi[r_{i+1}])[\rho] \wedge \neg\zeta[\rho]
\end{aligned}
$$

The deterministic symbolic synchronous product graph for two action systems $\mathcal{AS}_P$ and $\mathcal{AS}_S$ shall now be defined.

Definition 3.13 (Deterministic Product Graph).
Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \to_S \rangle$ be an action system representing a specification, let $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \to_P \rangle$ be an action system representing an implementation and let $d \in \mathbb{N}_0$ be the maximum exploration depth. The deterministic symbolic synchronous product graph $\mathcal{AS}_P \times_{sioco_{det}} \mathcal{AS}_S(d)$ bounded by d is a tuple $SP = \langle Q, q_{init}, \to_{SP}, \Lambda_I, \Lambda_U \rangle$ where $Q \subseteq \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_P} \times \mathbb{N}_0) \times \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S} \times \mathbb{N}_0)$, $\Lambda_\delta = \Lambda_I \cup \Lambda_U \cup \{\delta\}$, $\to_{SP} \subseteq Q \times \Lambda_\delta \times (Q \cup (\{fail\} \times \mathfrak{F}(\widehat{\mathcal{I}} \cup \mathcal{I})))$ and $q_{init} = (\tau_{cl}(\{(\top, \iota_P)_0\}), \tau_{cl}(\{(\top, \iota_S)_0\}))$. Transitions $(q, \lambda, q') \in \to_{SP}$ will be abbreviated by $q \xrightarrow{\lambda}_{SP} q'$. The transition relation $\to_{SP}$ and the set Q are defined as the smallest sets, satisfying the following rules:

Initial state:

$$\frac{}{q_{init} \in Q}$$

Execution of outputs:

$$\frac{\begin{array}{c}(\kappa_i, \mu_i) \in Q \qquad i < d \qquad \lambda \in \Lambda_U \qquad (\lambda, \varphi_P, \rho_P) \in \to_P \qquad (\lambda, \varphi_S, \rho_S) \in \to_S \\ \kappa_{i+1} = \tau_{cl}(exec(\kappa_i, \varphi_P, \rho_P)) \qquad \mu_{i+1} = \tau_{cl}(exec(\mu_i, \varphi_S, \rho_S)) \\ \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{i+1}} : \varsigma \models pc((\kappa_{i+1}, \mu_{i+1}))\end{array}}{(\kappa_{i+1}, \mu_{i+1}) \in Q \qquad (\kappa_i, \mu_i) \xrightarrow{\lambda}_{SP} (\kappa_{i+1}, \mu_{i+1})}$$

Observation of quiescence:

$$\frac{(\kappa_i, \mu_i) \in Q \qquad i < d \qquad \kappa_{i+1} = \tau_{cl}(exec(\kappa_i, \Delta_P, \mathsf{id})) \qquad \mu_{i+1} = \tau_{cl}(exec(\mu_i, \Delta_S, \mathsf{id})) \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{i+1}} : \varsigma \models pc((\kappa_{i+1}, \mu_{i+1}))}{(\kappa_{i+1}, \mu_{i+1}) \in Q \qquad (\kappa_i, \mu_i) \xrightarrow{\delta}_{SP} (\kappa_{i+1}, \mu_{i+1})}$$

Execution of inputs:

$$\frac{\begin{array}{c}(\kappa_i, \mu_i) \in Q \qquad i < d \qquad \lambda \in \Lambda_I \qquad (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \qquad (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \\ \zeta = \bigvee_{(\tau,\gamma,\pi) \in \rightarrow_P} \gamma \qquad \kappa_{i+1} = \tau_{cl}(exec(\kappa_i, \varphi_P, \rho_P) \cup exec_{neg}(\kappa_i, \varphi_P, \zeta)) \\ \mu_{i+1} = \tau_{cl}(exec(\mu_i, \varphi_S, \rho_S)) \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{i+1}} : \varsigma \models pc((\kappa_{i+1}, \mu_{i+1}))\end{array}}{(\kappa_{i+1}, \mu_{i+1}) \in Q \qquad (\kappa_i, \mu_i) \xrightarrow{\lambda}_{SP} (\kappa_{i+1}, \mu_{i+1})}$$

Detection of non-conformance:

$$\frac{\begin{array}{c}(\kappa_i, \mu_i) \in Q \qquad i \leq d \qquad \lambda \in \Lambda_U \cup \{\delta\} \qquad (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\} \\ (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\} \qquad (\chi, \eta)_i \in \mu_i \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_i \cup \mathcal{I}} : \varsigma \models \xi\end{array}}{(\kappa_i, \mu_i) \xrightarrow{\lambda} (fail, \xi)}$$

where $\xi = \left(\bigvee_{(\gamma_P, \pi_P) \in \kappa_i} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S, \pi_S) \in \mu_i} \gamma_S \wedge \varphi_S[\pi_S]\right)$

The implementation $\mathcal{AS}_P$ is not **sioco**$_d$-conform to $\mathcal{AS}_S$ iff there exists a path from q_{init} to a *fail*-state. Such a path combined with the condition associated with the corresponding *fail*-state forms a witness of non-conformance. Since **sioco** coincides with **ioco** [45] and the condition for non-conformance is based on the original definition of **sioco**, the product graph contains a *fail*-state iff there exists a suspension trace of $[\![\mathcal{AS}_S]\!]$ of length $\leq d$, which shows that $[\![\mathcal{AS}_P]\!]$ ~~**ioco**~~ $[\![\mathcal{AS}_S]\!]$.

There are a few things to note about the product graph. The condition which must be satisfied in order to be able to observe quiescence is denoted as Δ_P or Δ_S rather than Δ because the condition depends on whether the implementation or the specification is observed. Hence, Δ_P denotes that the condition is formed from transitions in $\rightarrow_P$, while Δ_S is derived via $\rightarrow_S$.

As noted before, action systems may behave non-deterministically. The **sioco** conformance relation, takes non-determinism into account as well. Therefore, the conformance check based on the deterministic product graph is able to handle action systems containing internal actions and does not produce spurious counterexamples. In other words, conforming action systems are not erroneously identified to be non-conforming.

Since mutated specification models are intended to be used as implementations, it is not possible to guarantee the input-enabledness of implementations, which is a requirement of the **sioco**-conformance relation. Hence, differently from the definition of **sioco** [45], implementations cannot be considered to be weakly input-enabled. An angelic completion is rather performed to make implementations input-enabled.

Furthermore, the product graph allows actions to be executed only if the path condition of the target product state, which contains both implementation and specification states, is satisfiable, while **ioco** is defined for suspension traces of the specification. This restriction is used as the non-conformance condition, given in Definition 3.14, would not be satisfiable anyway for product states with unsatisfiable path conditions. The non-conformance condition corresponds to the negation of the condition for **sioco**-conformance given by Frantzen et al. [45].

Example 3.3 (Product Graph for First-Order Mutant).
Let $\mathcal{ADD}$ be an action system defined as in Example 2.5. Consider a first-order mutant $\mathcal{ADD}_{mut}$ exactly equivalent to $\mathcal{ADD}$ except for the $?add$-transition. The guard of $?add$ shall be mutated by replacing $\leq$ for $<$. Hence, the $?add$-transition shall be defined by

$$(?add, par\_add\_p \geq 0 \wedge x + par\_add\_p < 50, \{x \mapsto x + par\_add\_p\}).$$

In order to distinguish the state variables of both action systems, x_S shall be used to refer to x of $\mathcal{ADD}$ and x_I shall be used to refer to x of $\mathcal{ADD}_{mut}$. To ease representation of formulas, p will denote the parameter of $?add$ instead of par_add_p.

The product graph $\mathcal{ADD}_{mut} \times_{sioco_{det}} \mathcal{ADD}(1)$ contains the initial state q_{init}, which is given by

$$q_{init} = (\{(\top, \{x_I \mapsto 0\})\}, \{(\top, \{x_S \mapsto 0\})\}).$$

Furthermore, it contains the edge $(q_{init}, ?add, q_1)$ connecting the initial state to one of its successors

$$\begin{aligned} q_1 =(\{\underbrace{(\neg(p_1 \geq 0 \wedge p_1 < 50), \{x_I \mapsto 0\})}_{angelic\ completion}, (p_1 \geq 0 \wedge p_1 < 50, \{x_I \mapsto p_1\})\}, \\ \{(p_1 \geq 0 \wedge p_1 \leq 50, \{x_S \mapsto p_1\})\}). \end{aligned}$$

Note that $?add$ leads to two different symbolic states in the implementation $\mathcal{ADD}_{mut}$ because angelic completion is applied since $?add$ is an input. As the product graph is bounded by 1, it may contain another edge from q_1 to a $fail$-state if the action systems are non-conforming.

It shall be checked if the $fail$-state reached by observing quiescence is part of the product graph, that is, it shall be checked if the corresponding non-conformance condition is actually satisfiable. The guard Δ of the quiescence observation is given by

$$\Delta = \neg\exists v : (\neg(x = 0) \wedge v = x) \Leftrightarrow x = 0.$$

The edge (q_1, δ, q_2) connects q_1 to the fail-state q_2 which is defined by

$$\begin{aligned} q_2 =(fail, ((\neg(p_1 \geq 0 \wedge p_1 < 50) \wedge (x_I = 0)[\{x_I \mapsto 0\}]) \\ \vee (p_1 \geq 0 \wedge p_1 < 50 \wedge (x_I = 0)[\{x_I \mapsto p_1\}])) \\ \wedge p_1 \geq 0 \wedge p_1 \leq 50 \wedge \neg(p_1 \geq 0 \wedge p_1 \leq 50 \wedge (x_S = 0)[\{x_S \mapsto p_1\}])). \end{aligned}$$

This state can be simplified to

$$q_2 = (fail, (p_1 < 0 \vee p_1 \geq 50 \vee p_1 = 0) \wedge p_1 \geq 0 \wedge p_1 \leq 50 \wedge p_1 \neq 0).$$

As the condition associated with q_2 is satisfied by the valuation $\{p_1 \mapsto 50\}$, the product graph contains the state q_2, which shows that $\mathcal{ADD}_{mut}$ does not conform to $\mathcal{ADD}$. The valuation $\{p_1 \mapsto 50\}$ is actually the only valuation satisfying the non-conformance condition. Furthermore, it detects the mutation as it corresponds to the only input accepted by the specification, but ignored by the implementation.

The discussed part of the product graph is depicted in Figure 3.2.

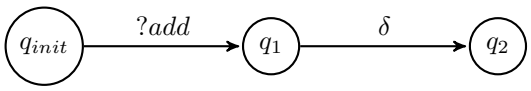


Figure 3.2: A part product graph $\mathcal{ADD}_{mut} \times_{sioco_{det}} \mathcal{ADD}(1)$ described in Example 3.3.

Non-conformance Condition for Product States

In the rule *Detection of non-conformance* of Definition 3.13, a condition for non-conformance is given directly without referring to a complex function like Φ, as in Definition 2.18. This is possible because this version of product graph is deterministic. The product graph is deterministic in the sense that given a start state and a sequence of observable actions, it is possible to uniquely determine the end state which is reached after executing the sequence. As a result, the states in the product graph are product states composed of compound symbolic states. These compound symbolic states consist of all satisfiable symbolic states reached after some trace is executed. This observation will be used for showing that Theorem 3.1 holds, that is, that the non-conformance condition for product states is equivalent to the condition given in Definition 3.2.

Definition 3.14 (Non-conformance Condition for Product States).
Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$ be an action system representing a specification and let $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$, be an action system representing an implementation. Let $d \in \mathbb{N}_0$ be the maximum exploration depth, and let $SP = \langle Q_{SP}, q_{init}, \rightarrow_{SP}, \Lambda_I, \Lambda_U \rangle$ be the deterministic symbolic synchronous product graph $\mathcal{AS}_P \times_{sioco_{det}} \mathcal{AS}_S(d)$. The non-conformance condition for a product state $(\kappa, \mu) \in Q_{SP}$ is given by:

$$\exists \lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} \left(\left(\bigvee_{(\gamma_P, \pi_P) \in \kappa} \gamma_P \wedge \varphi_P[\pi_P] \right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S, \pi_S) \in \mu} \gamma_S \wedge \varphi_S[\pi_S] \right) \right)$$
$$\textit{where } \exists \rho : (\chi, \rho) \in \mu$$
$$\textit{and } \exists \pi : (\lambda, \varphi_P, \pi) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\}$$
$$\textit{and } \exists \pi : (\lambda, \varphi_S, \pi) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\}$$

The non-conformance condition will also be used without explicit quantification of observations and parameter variables in the following. However, as the satisfiability of the unquantified formula will generally be examined, the parameter variables will implicitly be treated as existentially quantified. The unquantified formula will be referred to as non-conformance condition as well.

Theorem 3.1 (Equivalence of Non-conformance Conditions).
The non-conformance condition given in Definition 3.14 is equivalent to the non-conformance given in Definition 3.2. Hence, if (κ, μ) is a product state in a product graph SP, where SP, $\mathcal{AS}_S$, $\mathcal{AS}_P$, d are defined as in Definition 3.14 and Φ_{AS} is defined as in Definition 2.18, then there exists a (σ, χ) such that

$$\exists \lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} \left(\left(\bigvee_{(\gamma_P, \pi_P) \in \kappa} \gamma_P \wedge \varphi_P[\pi_P] \right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S, \pi_S) \in \mu} \gamma_S \wedge \varphi_S[\pi_S] \right) \right)$$
$$\Leftrightarrow \exists (\sigma, \chi) \in \mathcal{F}_S \; \exists \lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} \left(\Phi_P(\lambda, \sigma) \wedge \chi \wedge \neg \Phi_S(\lambda, \sigma) \right)$$
$$\textit{where } \exists \rho : (\chi, \rho) \in \mu$$
$$\textit{and } \mathcal{F}_S \textit{ is a set of initialised symbolic suspension traces of } \mathcal{AS}_S$$
$$\textit{and } \exists \pi : (\lambda, \varphi_P, \pi) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\}$$
$$\textit{and } \exists \pi : (\lambda, \varphi_S, \pi) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\}$$

Furthermore, for all initialised symbolic suspension traces (σ, χ), there exists a product state (κ, μ) such that the non-conformance conditions formed for both are equivalent.

In the following, Theorem 3.1 will be shown to hold. It will be shown that for all traces, there exists a product state such that the non-conformance conditions derived for both are equivalent. This corresponds to showing to completeness of the non-conformance check based on the product graph. Furthermore, it will be shown that for all product states (κ, μ), there exists a trace which reaches (κ, μ). This corresponds to showing soundness.

For the purpose of proving equivalence, the notion of traces shall be introduced for product graphs in a similar way as for LTSs by Tretmans [79]. Furthermore, $\mathcal{AS}_S$, $\mathcal{AS}_P$, d and SP will be assumed to be defined as in Definition 3.14.

Definition 3.15 (Traces in Product Graphs).
Let $\Lambda_\delta = \Lambda \cup \delta$ be a set containing action labels as well as δ and let Q_{SP} and $\rightarrow_{SP}$ be the states and the transition relation of product graph SP respectively.

$$
\begin{array}{ll}
q \overset{\sigma}{\Rightarrow} q & \textit{iff } \sigma = \epsilon \\
q \overset{\sigma_1 \cdot \ldots \sigma_n}{\Longrightarrow} q' & \textit{iff } \exists q_0, q_1, \ldots, q_n : q = q_0 \xrightarrow{\sigma_1}_{SP} q_1 \xrightarrow{\sigma_2}_{SP} \ldots \xrightarrow{\sigma_n}_{SP} q_n = q' \\
q \overset{\sigma_1 \cdot \ldots \sigma_n}{\Longrightarrow} & \textit{iff } \exists q' : q \overset{\sigma_1 \cdot \ldots \sigma_n}{\Longrightarrow} q'
\end{array}
$$

Lemma 3.1 (Relation between Product States and $\mathbf{after}_{\mathrm{s_{init}}}$).
Let $\sigma \in \Lambda_\delta^*$ be a sequence of actions and quiescence observations and let $(\kappa_i, \mu_i) \in Q_{SP}$ be an indexed product state contained in the deterministic product graph SP, such that $q_{init} \overset{\sigma}{\Rightarrow} (\kappa_i, \mu_i)$. It holds that:

$$
\begin{array}{rl}
\kappa_i = & \{(\varphi, \rho)_i \mid (\varphi, \rho)_i \in \mathbf{after}_{\mathrm{s_{init}}\, P}(\sigma, \top) \wedge \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi\} \\
\mu_i = & \{(\varphi, \rho)_i \mid (\varphi, \rho)_i \in \mathbf{after}_{\mathrm{s_{init}}\, S}(\sigma, \top) \wedge \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi\} \\
\textit{where} & \mathbf{after}_{\mathrm{s_{init}}\, P} \textit{ is the } after_{s_{init}}\textit{-function calculated for the implementation and} \\
& \mathbf{after}_{\mathrm{s_{init}}\, S} \textit{ is the } after_{s_{init}}\textit{-function calculated for the specification} \\
\textit{and} & length(\sigma) = i
\end{array}
$$

Lemma 3.1 states that a product state reached by a trace σ contains all satisfiable symbolic states reachable by executing σ from the initial states of the implementation and the specification. It follows from the following observations:

- The repeated application of τ_{cl}, $exec$ and $exec_{neg}$ for a sequence $\sigma \in \Lambda_\delta^*$ starting in the initial state of an action system, will create the same set of satisfiable symbolic states as the application of $\mathbf{after}_{\mathrm{s_{init}}}$ for $(\sigma, \top)$. While the definitions of τ_{cl}, $exec$ and $exec_{neg}$ contain satisfiability checks, the result of an application of $\mathbf{after}_{\mathrm{s_{init}}}$ may also contain additional unsatisfiable states.
- The application of $exec(\kappa, \phi, \rho)$ corresponds to the rule $\mathbf{S}\lambda$ defined for the symbolic suspension transition relation, where κ is a compound symbolic state and ϕ and ρ are the guard and the state update mapping of an observable action.
- The application of $exec(\kappa, \Delta, \mathsf{id})$ corresponds to the rule $\mathbf{S}\delta$ defined for the symbolic suspension transition relation, where κ is a compound symbolic state, Δ is the guard of the quiescence observation and id is the identity function, the state update mapping corresponding to quiescence.
- The definition of τ_{cl} corresponds to the rule $\mathbf{S}\tau$ defined for the symbolic suspension transition relation.
- Since the result of $\tau_{cl}(\{(\varphi, \rho)\})$, the τ-closure of (φ, ρ), also contains (φ, ρ), the rule $\mathbf{S}\epsilon$ defined for the symbolic suspension transition relation is also fulfilled by traces in the product graph.
- The application of $exec_{neg}(\kappa, \varphi, \zeta)$ corresponds to the rule $\mathbf{Sa}$ defined for the symbolic suspension transition relation, where κ is a compound symbolic state, φ is the guard of an input action and ζ is a disjunction over the guards of all internal actions.

Let σ and (κ_i, μ_i) be defined as in Lemma 3.1. Based on this lemma, it is possible to show that the detection of non-conformance is performed correctly. In other words, it is possible to show that checking the non-conformance condition for all product states is equivalent to checking the negation of the **sioco**-conformance condition. Let $(\lambda, \varphi, \rho) \in \rightarrow \cup \{(\delta, \Delta, \mathsf{id})\}$ with $\lambda \in \Lambda_U \cup \{\delta\}$ be an output

action or the quiescence observation and let (γ, π) be a symbolic state. The terms γ and $\varphi[\pi]$ correspond to the second and third tuple element of the symbolic observation of λ in state (γ, π). Since $|\{(\lambda, \varphi', \rho') \mid (\lambda, \varphi', \rho') \in \rightarrow\}| = 1$, it holds also that $|\{\gamma \wedge \varphi'[\pi] \mid (\lambda, \gamma, \varphi'[\pi]) \in \mathbf{out}_{\mathrm{s}}((\gamma, \pi))\}| = 1$. As a result, the formula $\bigvee\{\gamma \wedge \varphi'[\pi] \mid (\lambda, \gamma, \varphi'[\pi]) \in \mathbf{out}_{\mathrm{s}}((\gamma, \pi))\}$ can be rewritten to $\gamma \wedge \varphi[\pi]$, where $(\lambda, \gamma, \varphi[\pi]) \in \mathbf{out}_{\mathrm{s}}((\gamma, \pi))$ and φ is defined as above. Given the definition of $\mathbf{out}_{\mathrm{s}}$ for sets of states, the function Φ_P defined for the **sioco**-conformance condition in Definition 2.18 can be written as:

$$\Phi_P(\lambda, \sigma) = \bigvee \left\{ \phi \wedge \psi \;\middle|\; (\lambda, \phi, \psi) \in \bigcup_{(\gamma,\pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top)} \mathbf{out}_{\mathrm{s}}((\gamma, \pi)) \right\}$$

$$\Leftrightarrow \bigvee \left\{ \phi \wedge \psi \;\middle|\; (\lambda, \phi, \psi) \in \bigcup_{(\gamma,\pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top)} (\{(\lambda, \gamma, \varphi[\pi])\} \cup O) \right\}$$

where $\exists \rho : (\lambda, \varphi, \rho) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\}$ with $\lambda \in \Lambda_U \cup \{\delta\}$

and O is a set of symbolic observations with $\forall (\lambda', \psi', \rho') \in O : \lambda \neq \lambda'$

$$\Leftrightarrow \bigvee \left\{ \phi \wedge \psi \;\middle|\; (\lambda, \phi, \psi) \in \bigcup_{(\gamma,\pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top)} \{(\lambda, \gamma, \varphi[\pi])\} \right\}$$

$$\Leftrightarrow \bigvee_{(\gamma,\pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top)} \gamma \wedge \varphi[\pi]$$

$$\Leftrightarrow \left(\bigvee_{(\gamma,\pi) \in \kappa_i} \gamma \wedge \varphi[\pi] \right) \vee \left(\bigvee_{(\gamma,\pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top) \setminus \kappa_i} \gamma \wedge \varphi[\pi] \right)$$

since κ_i contains only satisfiable states:

$\gamma \leftrightarrow \bot$ for $\exists \pi : (\gamma, \pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top) \setminus \kappa_i$

$$\Leftrightarrow \left(\bigvee_{(\gamma,\pi) \in \kappa_i} \gamma \wedge \varphi[\pi] \right) \vee \left(\bigvee_{(\gamma,\pi) \in \mathbf{after}_{\mathrm{s}_{\mathrm{init}}}(\sigma, \top) \setminus \kappa_i} \bot \right)$$

$$\Leftrightarrow \bigvee_{(\gamma,\pi) \in \kappa_i} \gamma \wedge \varphi[\pi]$$

A similar form for the function Φ_S may be derived analogously:

$$\Phi_S(\lambda, \sigma) = \ldots \Leftrightarrow \bigvee_{(\gamma,\pi) \in \mu_i} \gamma \wedge \varphi[\pi]$$

where $\exists \rho : (\lambda, \varphi, \rho) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\}$ with $\lambda \in \Lambda_U \cup \{\delta\}$

The non-conformance condition for product states uses a formula χ as condition corresponding to trace σ, such that $\exists \rho : (\chi, \rho) \in \mu$ where μ is a compound symbolic state of the specification. It needs to be shown that $[\![(\sigma, \chi)]\!] \subseteq \mathit{Straces}(\mathsf{eval} \circ \iota_S)$. Since (χ, ρ) is a state of the specification, its interpretation $[\![(\chi, \rho)]\!]$ corresponds to a set of states $Q \subseteq \mathfrak{U}^{\mathcal{V}_S}$ in the IOLTS $[\![\mathcal{A}\mathcal{S}_S]\!]$. The set of suspension traces from the initial state $\mathsf{eval} \circ \iota_S$ to a state in $[\![(\chi, \rho)]\!]$ is given by $\{\sigma \in (\Sigma_I \cup \Sigma_U \cup \{\delta\})^* \mid \exists q' \in [\![(\chi, \rho)]\!] : \mathsf{eval} \circ \iota_S \stackrel{\sigma}{\Rightarrow} q'\}$, which is a subset of $\mathit{Straces}(\mathsf{eval} \circ \iota_S) = \{\sigma \in (\Sigma_I \cup \Sigma_U \cup \{\delta\})^* \mid \mathsf{eval} \circ \iota_S \stackrel{\sigma}{\Rightarrow}\}$. Hence, χ is a valid condition for an initialised symbolic suspension trace used for an **sioco**-conformance check. In other words, the non-conformance check based on the product graph is sound because it does not check for traces disallowed by the definition of **sioco**.

Let $(\kappa_i, \mu_i) \in Q_{SP}$ be a product state. The non-conformance condition given in Definition 3.2 can now be rewritten showing that both non-conformance conditions are equivalent.

$$\begin{aligned}
&\exists(\sigma,\chi) \in \mathcal{F}_S\ \exists\lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}}\cup\mathcal{I}}\,(\Phi_P(\lambda,\sigma) \wedge \chi \wedge \neg\Phi_S(\lambda,\sigma)) \\
&\qquad\qquad \text{where } \mathcal{F}_S \text{ is a set of initialised symbolic suspension traces} \\
&\qquad\qquad \text{such that } [\![\mathcal{F}_S]\!] = Straces(eval \circ \iota_S)_d \\
&\qquad \text{choose } \chi \text{ such that } \exists\rho : (\chi,\rho) \in \mu_i \\
&\qquad\quad \text{and } \sigma \text{ such that } q_{init} \overset{\sigma}{\Rightarrow} (\kappa_i, \mu_i) \\
&\qquad \exists\lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}}\cup\mathcal{I}} \Phi_P(\lambda,\sigma) \wedge \chi \wedge \neg\Phi_S(\lambda,\sigma) \\
&\Leftrightarrow \exists\lambda \in \Lambda_U \cup \{\delta\} \colon \overline{\exists}_{\widehat{\mathcal{I}}\cup\mathcal{I}} \left(\bigvee_{(\gamma,\pi)\in\kappa_i} \gamma \wedge \varphi_P[\pi] \right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma,\pi)\in\mu_i} \gamma \wedge \varphi_S[\pi] \right) \\
&\qquad\qquad \text{where } \exists\rho : (\chi,\rho) \in \mu_i \\
&\qquad\qquad\quad \text{and } \exists\rho_S : (\lambda, \varphi_S, \rho_P) \in \rightarrow_S \cup\{(\delta, \Delta_S, \mathsf{id})\} \\
&\qquad\qquad\quad \text{and } \exists\rho_P : (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup\{(\delta, \Delta_P, \mathsf{id})\}
\end{aligned}$$

Hence, if some initialised symbolic suspension trace (σ, χ) is chosen, a corresponding symbolic state of the specification reached by σ with path condition χ can be chosen. The non-conformance condition formed for (σ, χ) is semantically equivalent to the non-conformance condition formed for the product state reached by σ and the condition χ. Thus, the non-conformance check based on the product graph is also complete up to some bound, in addition to being sound. It follows that checking all conditions associated with *fail*-states in a product graph is equivalent to checking the negated conformance condition.

Difference to Simple Product Graph

Nevertheless, there is one difference compared to the simple product graph. The formula $\exists\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_{i+1}} : \varsigma \models \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{i+1}} \gamma_P\right) \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu_{i+1}} \gamma_S\right)$ is used in the rules to decide if some transition corresponding to an action should be contained in the product graph, which represents a satisfiability check of the path condition of the product state $(\kappa_{i+1}, \mu_{i+1})$. Thus, there may exist states (φ', ρ') in κ_{i+1} with path condition φ' such that $\varphi' \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu_{i+1}} \gamma'_S\right)$ is not satisfiable. These states would not be included in the simple product graph because the condition $\exists\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} \colon \varsigma \models \varphi'_P \wedge \varphi'_S$ is used for this graph, which is defined for single symbolic states. Consequently, the states (φ', ρ') could be filtered out in the second version of the product graph as well, but the conformance relation does not require to do so.

It shall now be proven that Proposition 3.1 holds. Hence, it shall be shown that the exclusion of product states with unsatisfiable path conditions is sound, as they are not relevant for the non-conformance check. This will be done by showing that a non-conformance formula formed for an unsatisfiable product state (κ, μ) is unsatisfiable as well.

Non-conformance Condition for Irrelevant Product States (κ, μ).

$$\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \underbrace{\neg\left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S \wedge \varphi_S[\pi_S]\right)}_{\xi}$$

where $\exists \rho_P : (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup\{(\delta, \Delta_P, \mathsf{id})\}$
and $\exists \rho_S : (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup\{(\delta, \Delta_S, \mathsf{id})\}$ with $\lambda \in \Lambda_U \cup \{\delta\}$
and $(\chi, \pi) \in \mu$
and $\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P\right) \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S\right) \leftrightarrow \bot$

The path condition of (κ, μ) shall be rewritten:

$$\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P\right) \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S\right) \leftrightarrow \bot$$

$$\Leftrightarrow \bigvee_{(\gamma_P,\pi_P)\in\kappa} \left(\gamma_P \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S\right)\right) \leftrightarrow \bot$$

$$\Rightarrow \forall \mu_{sub} \subseteq \mu : \bigvee_{(\gamma_P,\pi_P)\in\kappa} \left(\gamma_P \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu_{sub}} \gamma_S\right)\right) \leftrightarrow \bot$$

$$\Rightarrow \forall \mu_{sub} \subseteq \mu, \forall (\gamma_P, \pi_P) \in \kappa : \left(\gamma_P \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu_{sub}} \gamma_S\right)\right) \leftrightarrow \bot$$

Now the non-conformance condition shall be rewritten:

$$\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \xi$$

$$\Leftrightarrow \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P \wedge \chi \wedge \varphi_P[\pi_P]\right) \wedge \xi$$

$$\Leftrightarrow \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \left(\gamma_P \wedge \left(\bigvee_{(\gamma_S,\pi_S)\in\mu_{sub}} \gamma_S\right)\right) \wedge \varphi_P[\pi_P]\right) \wedge \xi$$

where $\mu_{sub} = \{(\chi, \pi)\} \subseteq \mu$

$$\Leftrightarrow \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \bot \wedge \varphi_P[\pi_P]\right) \wedge \xi \Leftrightarrow \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \bot\right) \wedge \xi \Leftrightarrow \bot \wedge \xi \Leftrightarrow \bot$$

□

By rewriting the path condition, it has been shown that the conjunction of the path condition of a single implementation state and the path condition of a subset of specification states is unsatisfiable if the corresponding product state is unsatisfiable. Such a conjunction can be formed by rewriting the non-conformance condition, where the subset of specification states has cardinality one. It follows that the non-conformance condition is unsatisfiable if the corresponding product state is unsatisfiable. Hence, Proposition 3.1 holds.

3.1.4 Unsafe States

Before the **sioco** checking algorithm can be presented, the notion of unsafe states needs to be introduced. A definition of unsafe states is given by Aichernig and Jöbstl [6] for a refinement-based approach to model-based mutation testing, which defines a state to be unsafe if an implementation may show non-conforming behaviour in the next step. Hence, in the context **sioco**, an unsafe state is a product state q such that the next observation may show **sioco** non-conformance. Stated differently, the pre-states of $fail$-states are unsafe. Formally, unsafe states can be defined as:

Definition 3.16 (Unsafe States).
Product states are unsafe if there exists a valuation of variables which satisfies the non-conformance condition given for the deterministic product graph and if they are reachable. Let $\mathcal{AS}_S$ and $\mathcal{AS}_P$ be action systems as defined in Definition 3.13, let $d \in \mathbb{N}_0$ and let q_{init} be the initial state of the deterministic symbolic synchronous product graph $\mathcal{AS}_P \times_{sioco_{det}} \mathcal{AS}_S(d)$. The set $Unsafe_d$, containing all unsafe states below depth d, is given by:

$$
\begin{aligned}
Unsafe_d = \Bigg\{ (\kappa_i, \mu_i) \;\Bigg|\; & (\kappa_i, \mu_i) \in \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_P} \times \mathbb{N}_0) \times \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S} \times \mathbb{N}_0) \wedge \\
& \exists \lambda \in \Lambda_U \cup \{\delta\}, \exists (\chi, \eta)_i \in \mu_i, \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}_i \cup \mathcal{I}} : \\
& \varsigma \models \left(\bigvee_{(\gamma_P, \pi_P) \in \kappa_i} \gamma_P \wedge \varphi_P[\pi_P] \right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S, \pi_S) \in \mu_i} \gamma_S \wedge \varphi_S[\pi_S] \right) \wedge \\
& \exists \sigma \in \Lambda_\delta^* : q_{init} \overset{\sigma}{\Rightarrow} (\kappa_i, \mu_i) \\
& \textit{where } (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\} \\
& \textit{and } (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\} \Bigg\}
\end{aligned}
$$

In the following, the depth bound d may be ignored if it is not relevant, thus the set $Unsafe$ may be considered which contains all unsafe states reachable through the execution of unbounded traces.

3.2 sioco Checking Algorithm

The **sioco** Checking algorithm implicitly explores the deterministic symbolic synchronous product graph defined in Section 3.1.3 and serves as the basis for the optimisations, which are introduced subsequently. More concretely, the basic version of the **sioco** checking algorithm shown in Algorithm 2 performs a bounded depth-first search for unsafe states in the deterministic product graph. If an unsafe state is found, a pair consisting of the satisfiable non-conformance condition and the trace leading to the unsafe state is returned, otherwise $conforming$ is returned. The latter signals that the implementation conforms to the specification up to a given depth. The following conventions will be used in the algorithm and in extensions presented afterwards in Chapter 4:

- The maximum search depth d is assumed to be globally accessible.
- An implementation $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$ will be checked for conformance to a specification $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$. The tuple elements of the implementation and the specification are assumed to be globally accessible.
- The union set of the transition relation and the set containing the quiescence observation shall be denoted by $\rightarrow_{\delta_P}$ and $\rightarrow_{\delta_S}$ respectively, where $\rightarrow_{\delta_{AS}} = \rightarrow_{AS} \cup \{(\delta, \Delta_{AS}, \mathsf{id})\}$.
- The empty trace shall be denoted by $[]$ and the $append$-function, which appends an action λ to a trace tr, shall be denoted by $tr \cdot \lambda$.

Additionally to the functions tau_{cl}, $exec$ and $exec_{neg}$ defined in the last section, the auxiliary functions defined below will be applied, with $\Lambda_{\tau\delta} = \Lambda \cup \{\tau\} \cup \{\delta\}$:

$$
\begin{aligned}
guard : \Lambda_{\tau\delta} \times (\Lambda_{\tau\delta} \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}}) &\to \mathfrak{F}(Var) \\
guard(\lambda, \to_{\delta_{AS}}) &\mapsto \bigvee_{(\lambda,\varphi,\rho)\in \to_{\delta_{AS}}} \varphi \\
update : \Lambda_{\delta} \times (\Lambda_{\tau\delta} \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}}) &\to \mathfrak{T}(Var)^{\mathcal{V}} \\
update(\lambda, \to_{\delta_{AS}}) &\mapsto \gamma \\
&\text{where } \{\gamma\} = \{\rho \mid \exists \varphi : (\lambda, \varphi, \rho) \in \to_{\delta_{AS}}\}
\end{aligned}
$$

Both functions, $guard$ and $update$, take the transition relation including the quiescence observation as second parameter. The function $guard$ additionally takes an action label and returns its guard if it corresponds to an observable action. However, if the action label is τ, the function returns the guards of all internal actions combined via disjunction. Conversely, the function $update$ is only defined for observable action labels and returns their state update mappings. Since there exists at most one transition for an observable action label, the state update mapping can be uniquely determined.

As noted above, Algorithm 2 performs a search for unsafe states in the bounded product graph via a depth-first strategy. The algorithm processes globally accessible data structures such as the transition relations of the considered action systems. At first (Lines 1 to 4), the search is initialised, thus the trace to the current state is set to be empty and the initial state $(initImpl, InitSpec)$ is set corresponding to q_{init} in Definition 3.13.

At each search step, non-conformance conditions for product states are checked (Lines 7 to 14). More specifically, they are checked for all path conditions χ associated with specification states (χ, ρ) (Line 7) and for all observations (Line 8). If the condition $nonConfCond$ is satisfiable, it is returned together with the trace of observable actions leading to the current state (Lines 10 to 12).

Otherwise, the search is continued if the maximum search depth has not been hit (Lines 15 to 31). Lines 16 to 22 basically execute each observable action simultaneously on the specification and on the implementation, while both action systems execute internal actions independently from each other by applying τ_{cl}. The current product state given by $(implState, specState)$ is thereby transformed into $(nextImpl, nextSpec)$. However, if some action is not executable by the specification (checked in Line 18), it is not executed by the implementation as well. Additionally, if an input action is executed, an angelic completion is performed for the implementation (Lines 20 to 22).

The search is continued if the reached product state $(nextImpl, nextSpec)$ is satisfiable, whereby the last executed action is appended to the trace leading to the current state (Lines 23 to 28).

After the execution of all observable actions, the constant value $conforming$ is returned to signal that non-conformance has not been detected. Since this algorithm returns witnesses of non-conformance $(nonConfCond, trace)$, it could be applied for test case generation, as these witnesses will be used for the actual testing of systems. However, it suffers from performance issues, which will be tackled in the following.

Algorithm 2 Basic version of the **sioco** checking algorithm.

1: **function** SIOCOCHECKINIT
2: $initSpec \leftarrow tau_{cl}(\{(\top, \iota_S)\})$
3: $initImpl \leftarrow tau_{cl}(\{(\top, \iota_P)\})$
4: **return** SIOCOCHECKREC([],$initSpec$,$initImpl$)
5: **end function**
6: **function** SIOCOCHECKREC($trace$,$specState$,$implState$)
7: **for all** $(\chi, \rho) \in specState$ **do**
8: **for all** $\lambda \in \Lambda_U \cup \{\delta\}$ **do**
9: $nonConfCond \leftarrow \begin{array}{l}\left(\bigvee_{(\gamma_P,\pi_P)\in implState} \gamma_P \wedge guard(\lambda, \rightarrow_{\delta_P})[\pi_P]\right) \wedge \chi \wedge \\ \neg\left(\bigvee_{(\gamma_S,\pi_S)\in specState} \gamma_S \wedge guard(\lambda, \rightarrow_{\delta_S})[\pi_S]\right)\end{array}$
10: **if** $nonConfCond$ is satisfiable **then**
11: **return** $(nonConfCond, trace)$
12: **end if**
13: **end for**
14: **end for**
15: **if** $length(trace) < d$ **then**
16: **for all** $\lambda \in \Lambda \cup \{\delta\}$ **do**
17: $nextSpec \leftarrow tau_{cl}(exec(specState, guard(\lambda, \rightarrow_{\delta_S}), update(\lambda, \rightarrow_{\delta_S})))$
18: **if** $|nextSpec| > 0$ **then**
19: $nextImpl \leftarrow tau_{cl}(exec(implState, guard(\lambda, \rightarrow_{\delta_P}), update(\lambda, \rightarrow_{\delta_P})))$
20: **if** $\lambda \in \Lambda_I$ **then**
21: $nextImpl \leftarrow \begin{array}{l}nextImpl \cup \\ \tau_{cl}(exec_{neg}(implState, guard(\lambda, \rightarrow_{\delta_P}), guard(\tau, \rightarrow_{\delta_P})))\end{array}$
22: **end if**
23: **if** $pc((nextImpl, nextSpec))$ is satisfiable **then**
24: $result \leftarrow$ SIOCOCHECKREC($trace \cdot \lambda$,$nextSpec$,$nextImpl$)
25: **if** $result \neq conforming$ **then**
26: **return** $result$
27: **end if**
28: **end if**
29: **end if**
30: **end for**
31: **end if**
32: **return** $conforming$
33: **end function**

4 Optimisations

A high-level description of the optimisations was presented at the USE-workshop [12].

The optimisations and extensions of the basic conformance checking algorithm developed in the course of this thesis shall now be presented. For this purpose, the working principle and the necessary adaptions of the algorithm will be discussed for each optimisation technique. The techniques utilise the previously discussed theoretical concepts and syntactic mutation analysis, which is inspired by the work of Aichernig and Jöbstl [6].

If necessary, additional functions and data structures will be introduced. As the goal is to implement a conformance check which is complete up to a given bound, proofs will be given showing that the optimisations do not alter the conformance checking results. In other words, it will be shown that a program applying the optimisations will produce the same results as the basic version of the algorithm. One optimisation, however, breaks this rule. Its application may lead to non-conformance conditions being undetected. This is done in order to improve the quality of generated test cases.

Since the complete version of the algorithm incorporating all optimisations is of complex structure, it will not be presented explicitly. A representation given in pseudo code would not facilitate understanding of the algorithm. Nevertheless, the conformance checker can be implemented based on the basic version of the algorithm and the presented optimisations.

4.1 Symbolic Execution Graph

The first technique is based on symbolic execution trees, which are discussed formally in Section 3.1.1. Since in model-based mutation testing, the conformance check between models is performed repeatedly using the same specification but with a large number of mutant implementations, precomputation can significantly increase the performance of the test case generation process. Hence, it makes sense to create a symbolic execution tree for the specification, which contains information about all executable paths of the system. This information can be used to improve the performance of the $exec$- and the τ_{reach}-functions for specifications. When these function are called during the conformance check, it is not necessary to perform satisfiability checks for the path conditions of the newly created symbolic states. It suffices to check if the nodes corresponding to the states are contained in the symbolic execution tree.

In general, the symbolic execution will suffer from a problem called path explosion for almost all non-trivial specifications [32], which is usually approached by discarding paths. There exist two groups of techniques to decide which paths can be discarded: heuristic techniques and sound program analysis. Since a complete conformance check shall be performed, a sound program analysis technique is used to prune paths of the symbolic execution tree. The idea behind the employed technique is similar to the idea behind the technique presented by Boonstoppel et al. [24]. A symbolic execution path can be pruned in a state, if an equivalent state was already explored, because the effects of further exploration are already known. More concretely, the set of enabled actions is already known, which is the information used in the conformance check.

The information is utilised as follows: the precomputed symbolic execution tree is explored during the conformance check until a state q is found, at which the tree was pruned. At this point, the exploration is carried on from the equivalent state, which was explored before q. Hence, the tree structure is actually a directed graph structure. Therefore, the term symbolic execution graph will be used to refer to a structure containing the pruned symbolic execution tree and state equivalence information.

The symbolic execution graph is created via Algorithm 3. This algorithm essentially creates a symbolic execution tree as defined in Section 3.1.1, but prunes it as described above. Furthermore, it creates a quotient set consisting of equivalence classes of symbolic states contained in the pruned tree. The quotient set corresponds to the set EQ in Definition 3.4. It may, however, contain less symbolic states than

EQ because of the pruning. Another result of pruning is that the symbolic execution graph is guaranteed to be of finite size, if the number of reachable concrete states and thereby also the number of equivalence classes is finite. Like the formally defined symbolic execution tree, the tree created by the algorithm also contains edges for the quiescence observation, as it is treated identically to ordinary actions. Algorithm 3 makes use of the following auxiliary functions and assumptions:

- The function $newId$ returns some unique identifier from a set ID, which is intended to be associated with a symbolic graph node.
- The function $emptyMap$ returns an empty mutable associative array. The function $put(M, k, v)$ adds a new value associated with the key k to the map M and $get(M, k)$ returns the value associated with key k. The function $keys(M)$ returns the set of keys for which there exists a key-value mapping in M.
- The function $emptyQueue$ returns an empty immutable queue. The functions $enqueue$, $dequeue$, $isEmpty$ and $peek$ are defined as usual.
- Sets are assumed to be mutable in Algorithm 3 and the function $add(S, v)$ adds a new element v to a set S.
- The function $actName$ maps a transition to its unique action name. This function is introduced because internal actions, like input and output action, also have unique names on the level of syntax. Hence, it is possible to distinguish τ-edges through this function, which enables optimisations based on a fine-grained syntactical analysis. The signature of $actName$ is

  $$actName : ((\Lambda \cup \{\tau\} \cup \{\delta\}) \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}}) \to ActionName,$$

  where $ActionName$ is a set of action names including the symbol δ, such that

  $$\forall n \in ActionName : |\{t \mid t \in \to_{\delta_S}, n = actName(t)\}| = 1 \text{ and}$$
  $$actName((\delta, \varphi, \rho)) = \delta$$

  are fulfilled. While the edges of the symbolic execution tree defined in Section 3.1.1 are labelled with elements of $\Lambda \cup \{\tau\} \cup \{\delta\}$, the edges of the symbolic execution graph created by Algorithm 3 are labelled with elements of $ActionName$.

The function $CreateTree$ defined in Algorithm 3 basically performs a breadth-first exploration of the symbolic execution tree. Lines 2 to 6 represent the initialisation of the algorithm: the root node $startNode$ is set, the quotient set $eqClasses$ is initialised to contain a set whose representative is the root node, the set of edges is initialised to be empty, and the queue Qs is initialised with the root node.

As long as there are graph nodes to be explored, one of the nodes is removed from Qs and processed (Lines 7 to 9). If the node was found at a depth i strictly lower than the maximum depth d, all actions are scheduled to be executed, while only internal actions need to be executed if the node was found at depth d (Lines 10 to 13). However, if the node was found beyond d, there is no need to explore it any further (Lines 14 and 15).

After determining which actions need to be executed, they are eventually executed (Line 18 to 22). Based on whether the currently executed action t is observable, indexes are added to parameters by applying r_{i+1} and the state index is increased. Furthermore, a new unique identifier is created via $newId()$. If the state $newQ$ reached by executing action t is satisfiable, an edge labelled with the name of t is added to the graph (Lines 23 and 24).

Pruning is performed in Lines 25 and 26 which check if a state equivalent to $newQ$ has already been explored. If there exists such a state, it is not necessary to explore $newQ$ any further and $newQ$ is added to an equivalence class. In other words, the symbolic execution graph is pruned at $newQ$. Otherwise, a new equivalence class containing only $newQ$ is created and added to $eqClasses$. Furthermore, $newQ$ is scheduled to be explored.

Algorithm 3 The symbolic execution graph creation algorithm.

1: **function** CREATETREE
2: $startNode \leftarrow ((\top, \iota_S)_0, newId())$
3: $eqClasses \leftarrow emptyMap$
4: $put(eqClasses, startNode, \{startNode\})$
5: $graphEdges = \{\}$
6: $Qs \leftarrow enqueue(emptyQueue, startNode)$
7: **while** $\neg isEmpty(Qs)$ **do**
8: $((\varphi, \rho)_i, id) \leftarrow peek(Qs)$
9: $Qs \leftarrow dequeue(Qs)$
10: **if** $i < d$ **then**
11: $actions \leftarrow \rightarrow_{\delta_S}$
12: **else if** $i = d$ **then**
13: $actions \leftarrow \{(\tau, \varphi, \rho) \mid (\tau, \varphi, \rho) \in \rightarrow_{\delta_S}\}$
14: **else**
15: $actions \leftarrow \{\}$
16: **end if**
17: **for all** $t = (\lambda, \psi, \pi) \in actions$ **do**
18: **if** $\lambda = \tau$ **then**
19: $newQ \leftarrow ((\varphi \wedge \psi[\rho], [\rho] \circ \pi)_i, newId())$
20: **else**
21: $newQ \leftarrow ((\varphi \wedge (\psi[r_{i+1}])[\rho], ([\rho] \circ ([r_{i+1}] \circ \pi))_{\mathcal{V}})_{i+1}, newId())$
22: **end if**
23: **if** $newQ$ is satisfiable **then**
24: $add(graphEdges, (((\varphi, \rho)_i, id), actName(t), newQ))$
25: **if** $\exists q \in keys(eqClasses) : q \equiv newQ$ **then**
26: $add(get(eqClasses, q), newQ)$ where $q \equiv newQ$
27: **else**
28: $put(eqClasses, newQ, \{newQ\})$
29: $Qs \leftarrow enqueue(Qs, newQ)$
30: **end if**
31: **end if**
32: **end for**
33: **end while**
34: **end function**

The data structures $graphEdges$, $eqClasses$ and the value $startNode$ are assumed to be globally accessible in the following. In order to use the data structures efficiently, the compound symbolic states of the specification need to be adapted. More concretely, each specification state processed in the **sioco** checking algorithm needs to be associated with a symbolic graph node. This is achieved by using pairs of symbolic states and graph node identifiers rather than symbolic states, which means that a specification state must be an element of the set $\mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S} \times \mathbb{N}_0) \times ID)$. Symbolic states in the symbolic execution graph are nodes, thus the terms nodes and states are used interchangeably in the following discussion. Hence, symbolic equivalence classes may also be formed for graph nodes.

Before it is possible to adapt $exec$ and τ_{reach}, another function rep shall be introduced. The function rep maps the identifier of a graph node n to the identifier of the canonical representative of the equivalence class to which n belongs. For an equivalence class C, the graph node $n \in C$ is chosen as canonical representative, which was explored first during the creation of the symbolic execution tree. This is done because the node explored first is also the only graph node in C having outgoing edges. Given the map $eqClasses$, the function rep can be defined as $rep(id) = r$ where r is chosen such that

$\exists(id', s) \in get(eqClasses, r) : id' = id$ is fulfilled. If the number of graph nodes is large, the function rep should be precomputed and for instance stored in a hash table. By means of this function and given the data structures created via Algorithm 3, it is now possible to change the definition of $exec$ and τ_{reach}. For simplicity of representation, symbolic state indices will be ignored for the definition of $\tau_{reach_{graph}}$, but it can trivially be extended to account for indices.

$$\tau_{reach_{graph}} : \mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S}) \times ID) \to \mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S}) \times ID)$$

$$\tau_{reach_{graph}} : S \mapsto \bigcup_{((\varphi,\rho),id)\in S} \Big\{((\varphi \wedge \psi[\rho], [\rho] \circ \pi), rep(id')) \;\Big|\; (\tau, \psi, \pi) \in \to_{\delta_S} \wedge \exists q, q' : ((q, id), actName((\tau, \psi, \pi)), (q', id')) \in graphEdges\Big\}$$

$$exec_{graph} : \mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S} \times \mathbb{N}_0) \times ID) \times \Lambda_\delta \to \mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S} \times \mathbb{N}_0) \times ID)$$

$$exec_{graph} : (\kappa, \lambda) \mapsto \Big\{((\varphi \wedge (\psi[r_{i+1}])[\rho], ([\rho] \circ ([r_{i+1}] \circ \pi))\mathcal{V}_S)_{i+1}, rep(id')) \;\Big|\; (\lambda, \psi, \pi) \in \to_{\delta_S} \wedge ((\varphi, \rho)_i, id) \in \kappa \wedge \exists q, q' : ((q, id), actName((\lambda, \psi, \pi)), (q', id')) \in graphEdges\Big\}$$

The identifier id is used to guide the search through the symbolic execution graph. It allows to find edges in the symbolic execution graph. Compared to the definition of $exec$, $exec_{graph}$ takes the label of an action as parameter rather than guard and state update because it is needed to query the symbolic execution graph. The guard and state update are retrieved from the globally accessible transition relation.

It should be noted that given a symbolic state $(\varphi, \rho)_i$ and an associated graph node $((\psi, \pi)_j, id)$, the symbolic states $(\varphi, \rho)_i$ and $(\psi, \pi)_j$ are not interchangeable in the context of the conformance check, although $(\varphi, \rho)_i \equiv (\psi, \pi)_j$. In other words,$((\psi, \pi)_j, id)$ must not be used for purposes other than applying $exec_{graph}$ and $\tau_{reach_{graph}}$ because $\mathsf{free}(\varphi)$ is not necessarily equal to $\mathsf{free}(\psi)$. Considering the simultaneous execution and the non-conformance condition, the implementation action system $\mathcal{AS}_P$ places additional constraints on the variables in $\mathsf{free}(\varphi)$. Hence, the non-conformance condition would be formed incorrectly if $(\psi, \pi)_j$ was used instead of $(\varphi, \rho)_i$.

By applying this optimisation, it is possible to significantly decrease the number of satisfiability checks needed for the execution of actions. There are two reason for this reduction. The first one is that it is not necessary to perform satisfiability checks for the specification during the conformance check because they have already been performed when the symbolic execution graph was created. The second reason is that the pruning further reduces number of satisfiability checks needed. This pruning is effective because reactive systems generally show looping behaviour [47], that is, they come back to already visited states.

Nevertheless, equivalence checks may also lead to poor performance, if the involved path conditions and state vectors contain complex terms. In cases where only low exploration depths are needed, it may be more efficient to explore redundant paths than to perform equivalence checks. Poor performance of equivalence checks will be addressed in Section 7.4. However, in most cases, it is beneficial or even necessary to prune search paths because of the aforementioned path explosion problem.

4.2 Product Graph Pruning

The optimisation-technique *product graph pruning* shares some similarities with the *symbolic execution graph* optimisation-technique. Similarly, this technique also tackles the path explosion problem. The problem affects the product graph as well because the product graph is explored for all bounded traces of the specification. Hence, it is actually necessary to fully explore a bounded symbolic execution tree if the implementation conforms to the specification. Strictly speaking, a full exploration may not be needed

if not all traces are executable by the implementation but the conformance check would still suffer from path explosion.

Moreover, a full exploration would render the pruning of the symbolic execution graph useless, as it would be necessary to unfold the loops in the graph anyway. However, it is possible to prune the product graph following an approach based on equivalence checks as well.

A product state is a pair of compound symbolic states, thus a product state examined in Algorithm 2 would be the pair $(implState, specState)$. To prune the product graph, it is necessary to keep track of all indexed product states visited during the conformance check. Given this information, a set of visited states V, it is possible to improve the performance of the conformance check in two ways:

- If V contains a product state q, such that $q \equiv_{prod} (implState, specState)$, it is not necessary to check if the non-conformance condition is satisfiable in state $(implState, specState)$. This does not actually prune the product graph because the product graph does not contain edges for unsatisfiable non-conformance conditions. Nevertheless, it decreases the number of necessary satisfiability checks. An algorithm applying this optimisation will also perform a complete **sioco** check because if the current state would be an unsafe state, its equivalent state q would have already been identified as unsafe. Consequently, the search would have terminated before reaching the current product state. Hence, it is not necessary to check the non-conformance condition.
- If V contains a product state q with index i such that $q \equiv_{prod} (implState, specState)$ and $i \leq j$, where j is the current search depth, then the search can be stopped. Stopping the exploration of the product graph corresponds to pruning of the product graph. The idea behind this optimisation is that the current product state fully determines, which states are explored at subsequent steps and that the exploration of equivalent states leads to equivalent post-states. Hence, if an unsafe state would be reached after visiting $(implState, specState)$, an unsafe state would also be reached after visiting q. Conversely, if q does not lead to unsafe states, then $(implState, specState)$ does not lead to unsafe states as well.

These optimisations rely on the assumption that for equivalent product states q and q', $q \notin Unsafe$ implies $q' \notin Unsafe$. This fact is easier to show on a concrete rather than on a symbolic level. Hence, the relationship between **sioco** and **ioco** and a characterisation of unsafe IOLTS-product states shall be utilised.

Definition 4.1 (Unsafe States - IOLTS).
Let $S_1 = \langle Q_1, s_{0_1}, \Sigma_I, \Sigma_U, \rightarrow_1 \rangle$ and $S_2 = \langle Q_2, s_{0_2}, \Sigma_I, \Sigma_U, \rightarrow_2 \rangle$ be two IOLTS and let $SP = S_1 \times_{ioco} S_2$ be the synchronous product defined by the rules given by Weiglhofer and Wotawa [83], such that SP is an IOLTS $SP = \langle Q_{SP}, s_{0_{SP}}, \Sigma_I, \Sigma_U, \rightarrow_{SP} \rangle$. The set of unsafe product states $Unsafe_{ioco}$, where $Unsafe_{ioco} \subseteq Q_1 \times Q_2$, can be defined as follows:

$$Unsafe_{ioco} = \left\{ (s_1, s_2) \;\middle|\; s_1 \in Q_1 \wedge s_2 \in Q_2 \wedge \exists \lambda \in \Lambda_U \cup \{\delta\} : s_1 \xrightarrow{\lambda}_1 \wedge s_2 \not\xrightarrow{\lambda}_2 \right\}$$

or equivalently

$$Unsafe_{ioco} = \{ (s_1, s_2) \mid s_1 \in Q_1 \wedge s_2 \in Q_2 \wedge out(s_1) \not\subseteq out(s_2) \}$$

Hence, an unsafe state in the synchronous product $S_1 \times_{ioco} S_2$ is a state s such that there exists an edge from s to a *fail*-state, that is, the next step may show **ioco** non-conformance.

Since Frantzen et al. have shown that **sioco** coincides with **ioco** [45] and the definitions of the sets $Unsafe$ and $Unsafe_{ioco}$ are based on **sioco** non-conformance and on **ioco** non-conformance respectively, it can be concluded that the interpretation of an unsafe symbolic product state contains at least one unsafe IOLTS-product state. Conversely, if a product state does not show **sioco** non-conformance and is thus not unsafe, its interpretation must not show non-conformance as well.

It follows that if a symbolic product state q is not unsafe then it holds that

$$[\![q]\!] \cap Unsafe_{ioco} = \emptyset.$$

For a product state q' equivalent to q it holds that

$$[\![q]\!] = [\![q']\!], \text{ thus } [\![q']\!] \cap Unsafe_{ioco} = \emptyset$$

is also fulfilled. As $[\![q']\!]$ does not contain unsafe IOLTS-product states and **sioco** coincides with **ioco**, it follows that q' is not unsafe. Hence, the implication

$$q \notin Unsafe \Rightarrow q' \notin Unsafe$$

holds if $q \equiv q'$.

Since the optimisations presented above rely on equivalence checks, they may also lead to poor performance if the formulas, which need to be checked, contain complex terms. As a result, the same holds as for the first optimisation and it may be more efficient to perform the conformance check without applying this technique. It should be noted though that pruning is beneficial in general and should be considered.

4.3 Syntactic Mutation Analysis

Before discussing the optimisations made possible by a mutation analysis on syntax-level, the mutation analysis itself and its results shall be described. This precomputation step is performed before the start of the conformance check and assumes that the implementation is a first-order mutant of the specification. Hence, the optimisations based on syntactic mutation analysis are only applicable for a limited set of implementations.

Additionally, first-order mutants must adhere to the following limitations:

- type-definitions must not be mutated
- the types of state variables must not be mutated
- the types and number of action parameters must not be mutated
- actions must not be added or deleted
- the types of actions must not be mutated

The first two restrictions could be loosened. The other three restrictions, however, must be fulfilled for the **sioco** check to be applicable because the **sioco** conformance relation requires that implementation and specification must share the same set of actions and parameter variables. This requirement also demands that an output action of the specification must not be an input action of the implementation and thus action types may not be mutated as well.

The syntactic mutation analysis compares an implementation with the specification by performing Algorithm 4. It uses the set $ActionName$ and the function $actName$ introduced before. $ActionName$ contains all action names and $actName$ maps transitions to action names. The function $getInitBlock$ retrieves the syntactical representation of the init block. Furthermore, the functions $getAction(AS, a)$ and $getGuard(AS, a)$ retrieve the syntactical representation of an action a and the guard of an action a of the action system AS.

Algorithm 4 Syntactic mutation analysis algorithm.

```
1:  procedure SYNTMUTANALYSIS(specification,implementation)
2:      if getInitBlock(specification) ≠ getInitBlock(implementation) then
3:          mutAction ← Init
4:          intMut ← False
5:          intOrOutMut ← False
6:          intOrOutGuardMut ← False
7:      else
8:          for all a ∈ ActionName where a ≠ δ do
9:              if getAction(specification, a) ≠ getAction(implementation, a) then
10:                 (λ, φ_S, ρ_S) ← t_S such that t_S ∈ →_{δ_S} and actName(t_S) = a
11:                 (λ, φ_P, ρ_P) ← t_P such that t_P ∈ →_{δ_P} and actName(t_P) = a
12:                 mutAction ← a
13:                 intMut ← λ = τ
14:                 intOrOutMut ← intMut ∨ λ ∈ Λ_U
15:                 intOrOutGuardMut ← intOrOutMut ∧ getGuard(specification, a) ≠
                    getGuard(implementation, a)
16:                 break
17:             end if
18:         end for
19:     end if
20: end procedure
```

The algorithm sets the globally accessible variables described below:

$intMut$**:** signals whether an internal action was mutated

$intOrOutMut$**:** signals whether an internal or an output action was mutated

$intOrOutGuardMut$**:** signals whether the guard of an internal or an output action was mutated

$mutAction$**:** holds the name of the mutated action or the value $Init$, which is chosen such that it is unequal to all action names

As the procedure is intended to be applied for first-order mutants, it will only look for exactly one difference between specification and implementation. Consequently, if a difference is found in the *init*-block, there is no need to compare actions. If a difference is found for one action, the search can be stopped as well. The procedure in Algorithm 4 does not retrieve all information about mutations, but rather focuses relevant information. As a result, it does not check state updates.

The information about mutations can be used as the basis for further optimisations. Intuitively, the effects of executing first-order mutants will largely be the same as the effects of executing the specification because first-order mutants differ from their unmutated specification in only one aspect. This observation is the key insight forming the basis of various optimisations and the discussion in the following sections will elaborate on it.

4.4 Restriction of Angelic Completion for Mutants

As indicated above, a mutated specification serving as implementation can be assumed to behave similarly to the specification. Moreover, a first-order mutant may be assumed to behave identically to the specification as long as the mutated action has not been performed. Since implementations are considered to be input-enabled, this is not entirely true. The angelic completion of an action system may behave differently than the original action system. Hence, an input-enabled mutant may behave differently than the

specification even if the unmutated action has not been performed yet. Consider the following example showing that angelic completion may lead to **sioco** non-conformance.

Example 4.1 (Non-conformance through Angelic Completion).
Given is an action system $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ *using integers as data, where:*

- $\mathcal{V} = \{s\}$,
- $\mathcal{I} = \{\}$,
- $\Lambda_I = \{?x\}$,
- $\Lambda_U = \{!a\}$,
- $\iota = \{s \mapsto 0\}$ *and*
- $\rightarrow = \{(\tau, s = 0, \{s \mapsto 1\}), (\tau, s = 0, \{s \mapsto 2\}),$
  $(?x, s = 1, \{s \mapsto 3\}), (!a, s = 2, \{s \mapsto 4\})\}$.

This action system shall serve both as specification and as implementation and a non-conformance check after executing the trace $?x$ *shall performed. The initial product state is given by* (κ_0, κ_0)*, where*

$$\kappa_0 = \{(\top, \{s \mapsto 0\}), (\top, \{s \mapsto 1\}), (\top, \{s \mapsto 2\})\}$$

and the product state after executing $?x$ *is given by* (κ_1, μ_1)*, where*

$$\mu_1 = \tau_{cl}(exec(\kappa_0, s = 1, \{s \mapsto 3\})) = \{(\top, s \mapsto 3)\}$$

and

$$\begin{aligned} \kappa_1 &= \tau_{cl}(exec(\kappa_0, s = 1, \{s \mapsto 3\}) \cup exec_{neg}(\kappa_0, s = 1, s = 0)) \\ &= \{(\top, s \mapsto 3), (\top, s \mapsto 2)\}. \end{aligned}$$

The non-conformance condition for action $!a$ *is satisfiable as is shown below.*

$$\underbrace{\left(\left(\underbrace{\top}_{pathcondition} \wedge \underbrace{2 = 2}_{guard}\right) \vee \left(\underbrace{\top}_{pathcondition} \wedge \underbrace{2 = 3}_{guard}\right)\right)}_{implementation} \wedge \top \wedge \neg \underbrace{\left(\underbrace{\top}_{pathcondition} \wedge \underbrace{2 = 3}_{guard}\right)}_{specification} \Leftrightarrow \top$$

This behaviour is not the result of a wrong definition of angelic completion of action systems, but can also be observed during **ioco** conformance checking. More specifically, the angelic completion of the interpretation $[\![\mathcal{AS}]\!]$ is not **ioco** conform to $[\![\mathcal{AS}]\!]$. Both the IOLTS-interpretation $[\![\mathcal{AS}]\!]$ and its angelic completion are shown in Figure 4.1[1]. A comparison of the sets of observations possible after executing $?x$ shows that **ioco** non-conformance can also be detected in the IOLTS-models. The observations produced by the angelic completion are given by

$$out(i_0 \textbf{ after } ?x) = \{\delta, !a\}$$

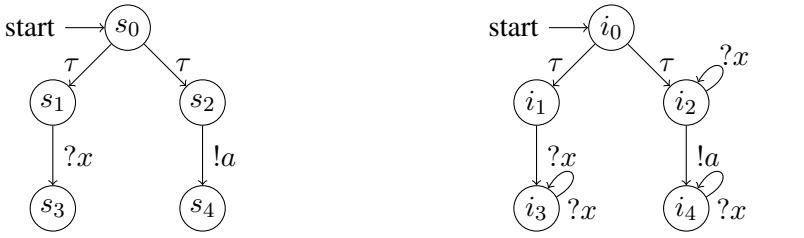
and the observations produced by the specification after executing $?x$ are given by

$$out(s_0 \textbf{ after } ?x) = \{\delta\}.$$

Since $\{\delta, !a\} \not\subseteq \{\delta\}$, the implementation does not conform to the specification from which it was derived via angelic completion.

Hence, angelic completion can actually be seen as a mutation, thus an **sioco** conformance-check for a first order mutant would actually be a check for a higher-order mutant, an action system mutated more than once. This is contradictory to the intention of employing first-order mutants as the basis for test

[1] A similar example is used by Aichernig et al. to discuss controllability issues [3].



(a) specification **(b)** input-enabled implementation

Figure 4.1: The IOLTS-interpretation of the action system given in Example 4.1 shown on the left and its angelic completion shown on the right.

case generation. Moreover, if a conformance violation due to angelic completion is detected after a low number of steps during the conformance check, then it may be detected for a large number of mutants before their mutated actions can be executed. This would in turn result in a large number of equivalent test cases, which is not desirable. The goal is actually to find a great variety of test cases covering different faults. To achieve this and counter the aforementioned problem, the angelic completion shall not be performed before the mutated action is executed. Obviously, this will result in an incomplete check, but will lead to the generation of more effective tests. Furthermore, it will only affect model-based mutation testing.

An intuitive argument justifying this decision is that implementation and specification will execute the same internal actions as long as the mutated action has not been executed because they will behave the same until the mutation takes effect, making angelic completion unnecessary. As a result, the state of the mutant implementation and the specification will also be the same until the mutated action is executed.

4.5 Avoiding the Execution of Implementation Actions

This optimisation avoids performing satisfiability checks for the execution of actions of the implementation as long as possible. These checks need not be performed as long as the mutated action has not been executed and quiescence has not been observed because the implementation state will be the same as the specification state until the mutation takes effect. However, there are certain pitfalls to avoid, so the following observations shall outline how to implement this optimisation. The first seven observations concern deterministic action systems, which are action systems not containing internal actions.

1. If the *init*-block is mutated, this optimisation can not be applied and thus satisfiability checks for the execution of all implementation actions need to be performed.
2. If the mutated action has not been executed yet and the action performed next is not mutated, then the next implementation state will be the same as the next specification state. Consequently, it is not necessary to execute the action for the implementation. Hence, the next implementation state can be set to be equal to the next specification state. As noted before, it is not necessary to perform $exec_{neg}$ as well because the implementation is assumed to be behave identically to the specification and therefore will accept the same set of inputs.
3. If the mutated action has not been executed yet and it is the action performed next, then the next implementation state may be different from the next specification state, thus making it necessary to execute the action for the implementation.
4. If the mutated action has not been executed yet, and the mutation affects the guard of an output action, and quiescence is observed next, then the next implementation state may be different from the next specification state. This is caused by the dependence of the quiescence condition on

the guards of outputs. Hence, it is necessary to perform satisfiability checks of the quiescence condition for the implementation and the specification separately.

5. If the mutated action has not been executed yet, and the mutation does not affect the quiescence condition, and quiescence is observed next, then the next implementation state will be the same as the next specification state.
6. If the mutated action has been executed, all subsequent actions and quiescence observations need to be performed for the implementation.
7. If quiescence has been observed and the mutation affects the guard of an output, all subsequent actions and quiescence observations need to be performed for the implementation.

The next observations are concerned with action systems containing internal actions.

8. If the mutated action is not an internal action, the same observations as for deterministic systems need to be considered.
9. If action systems contain internal actions, there may be several different sequences of actions leading to some compound symbolic state. Consider the case where an internal action is mutated: there may be sequences of actions executed on the implementation, which do not contain the mutated action while other observably equivalent sequences contain the mutated action.

   Hence, the compound implementation state may contain symbolic states which are affected by the mutation and some which are not affected. As a result, it may contain symbolic states equivalent to symbolic states contained in the compound specification state, although both compound states are not syntactically equivalent.

   If actions are executed for symbolic states reached by sequences not containing the mutated internal action, the information provided by the symbolic execution graph may be utilised. This is possible as the concerned states are equivalent to symbolic states of the specification. Consequently, symbolic states of the implementation shall also be associated with symbolic graph nodes.
10. If the mutated action is an internal action and its guard is mutated then all τ-closures need to be calculated explicitly without utilising the symbolic graph, because the mutated internal action may not be enabled for the same set of symbolic states as the unmutated internal action. Hence, if the mutated action is executed during the calculation of the τ-closure q, the states in q must not be associated with symbolic graph nodes, as they are in general not equivalent to states of the specification.

    If the τ-closure is calculated for an implementation state associated with a symbolic graph node and the mutated action is not executed during the calculation of the τ-closure, the τ-closure can be calculated again via the function $\tau_{reach_{graph}}$. Since the τ-closure can be calculated very efficiently using the symbolic execution graph, the additional computational effort pays off because subsequent operations can make use of the symbolic execution graph.
11. If the mutated action is an internal action and its body is mutated then it is possible to calculate the τ-closure by utilising the symbolic execution graph unless the mutated action is reachable from the current state. If the mutated action is reachable, the τ-closure needs to be calculated explicitly.
12. If the mutated action has not been executed before, and the mutation affects the guard of an internal action, and quiescence is observed next, then the next implementation state may be different from the next specification state. The reason for this is that the quiescence condition also depends on the guards of internal actions. As a result, it is necessary to perform checks of the quiescence condition for the implementation and the specification separately.
13. If quiescence has been observed and the mutation affects the guard of an internal action, then all subsequent actions and quiescence observations need to be performed for the implementation.

In summary, the symbolic states of the implementation shall be associated with symbolic graph nodes. This enables the utilisation of the symbolic execution graph as long as the mutated action is not executed and as long as quiescence is not observed. If the mutated action is an observable action, it is not necessary to perform any operation for the implementation as long as the mutation does not take effect because the states of the specification and the implementation are equivalent until then. Special care has to be taken if the mutation affects the guards of internal actions or output actions as the quiescence observation depends on those.

4.6 Simplifying Equivalence Checks for Product States

The following optimisation can be seen as an extension of the last extension. Since the state of the implementation is syntactically equivalent to the state of the specification until the mutation takes effect, the product state equivalence check can be simplified and thereby optimised as well. A product state consisting of two syntactically equivalent compound symbolic states has the form (κ, κ). It is equivalent to a product state (μ, μ) of the same form if $\kappa \equiv_{com} \mu$. Consequently, the product state equivalence check can be substituted for a compound symbolic state equivalence check. However, this check can be further simplified by considering the symbolic graph nodes associated with the symbolic states in the compound symbolic states. Using the symbolic state equivalence classes calculated by Algorithm 3, it is possible to formulate a sufficient condition for equivalence of compound symbolic states. This condition can be checked more efficiently than the condition given by Definition 3.9. For the derivation of the condition the function $state$ shall be used, which maps a graph node identifier to the corresponding symbolic state. It is defined as

$$\begin{aligned} state(id) =& (\varphi, \rho)_i \text{ where } (\varphi, \rho)_i \text{ is chosen such that} \\ & ((\varphi, \rho)_i, id) \in get(eqClasses, rep(id)) \text{ is fulfilled.} \end{aligned}$$

In the following, it will be shown that equivalence of representatives in two compound symbolic states s_1 and s_2 implies $s_1 \equiv_{com} s_2$:

Let s_1 and s_2 be two compound symbolic states associated with symbolic graph nodes,
that is $s_i \in \mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \times ID)$
define $R(s_i) = \{rep(id) \mid \exists q : (q, id) \in s_i\}$ as the set of representatives.

$R(s_1) = R(s_2) \Leftrightarrow$
$\{rep(id) \mid \exists q : (q, id) \in s_1\} = \{rep(id) \mid \exists q : (q, id) \in s_2\} \Leftrightarrow$
$\{state(rep(id)) \mid \exists q : (q, id) \in s_1\} = \{state(rep(id)) \mid \exists q : (q, id) \in s_2\} \Leftrightarrow$
$\{[\![state(rep(id))]\!] \mid \exists q : (q, id) \in s_1\} = \{[\![state(rep(id))]\!] \mid \exists q : (q, id) \in s_2\} \Leftrightarrow$
$q = state(id)$ for $(q, id) \in s_i$ and $state(id) \equiv state(rep(id))$
implies $q \equiv state(rep(id)) \Leftrightarrow [\![q]\!] = [\![state(rep(id))]\!]$
$\{[\![q]\!] \mid \exists id : (q, id) \in s_1\} = \{[\![q]\!] \mid \exists id : (q, id) \in s_2\} \Rightarrow$
$\bigcup\{[\![q]\!] \mid \exists id : (q, id) \in s_1\} = \bigcup\{[\![q]\!] \mid \exists id : (q, id) \in s_2\} \Rightarrow$
$[\![\{q \mid \exists id : (q, id) \in s_1\}]\!] = [\![\{q \mid \exists id : (q, id) \in s_2\}]\!] \Leftrightarrow$
definition of compound state equivalence
$\{q \mid \exists id : (q, id) \in s_1\} \equiv_{com} \{q \mid \exists id : (q, id) \in s_2\} \Leftrightarrow$
$s_1 \equiv_{com} s_2$

Hence, two compound symbolic states are equivalent if their sets of representatives are equal. Although this is not a necessary condition, it shall be used as an approximation, as the exact equivalence check is computationally expensive.

Another approximation shall be introduced for the general case, in which a state is reached after executing the mutated action. Let (κ, μ) and (η, ω) be two product states. In most cases where the specification states μ and ω are not equivalent, (κ, μ) will not be equivalent to (η, ω) as well. Consequently, $(\kappa, \mu) \equiv_{prod} (\eta, \omega)$ shall be approximated by $\mu \equiv_{com} \omega \wedge (\kappa, \mu) \equiv_{prod} (\eta, \omega)$, such that the right operand of the conjunction is only evaluated if the left operand evaluates to true. Since the compound symbolic states μ and ω are states of the specification, they are associated with symbolic graph nodes, thus the condition may be approximated further by $R(\mu) = R(\omega) \wedge (\kappa, \mu) \equiv_{prod} (\eta, \omega)$.

In order to be able to use both approximations, the information about visited states shall be extended. For each visited product state p a pair $(p, equal)$ shall be saved, where $equal \in \{True, False\}$ is set to $True$ if p was reached without executing the mutated action and set to $False$ otherwise. Stated differently, $equal$ denotes whether p contains two syntactically equivalent compound symbolic states. The *init*-block is assumed to be the first executed action, which means that $equal$ will be set to $False$ for all visited product states if *init* is mutated. It is now possible to define an equivalence condition approximation for product states extended with Boolean equivalence flags.

Definition 4.2 (Product State Equivalence - Approximation).
Let $((\kappa, \mu), e)$ and $((\kappa', \mu'), e')$ be two product states associated with Boolean values e and e' denoting whether they consist of syntactically equivalent compound symbolic states

$$((\kappa, \mu), e) \equiv_{approx} ((\kappa', \mu'), e') = \begin{cases} R(\mu) = R(\mu') & \text{if } e = True \wedge e' = True \\ \bot & \text{if } e = True \wedge e' = False \\ (\kappa, \mu) \equiv_{prod} (\kappa', \mu') & \text{if } e = False \wedge R(\mu) = R(\mu') \\ \bot & \text{if } e = False \wedge R(\mu) \neq R(\mu') \end{cases}$$

Obviously, $\equiv_{approx}$ is not an equivalence relation because it is not symmetric. Nevertheless, this does not introduce incorrect behaviour as $\equiv_{approx}$ is only checked to decide if product graph traces should be pruned. It is not used to calculate equivalence classes. The condition is intended to be checked such that the current product state is the left operand and some visited state is the right operand.

Since the second case has not been discussed above, it shall be shortly investigated. It identifies the current product state to be inequivalent to a product state q if the current state was reached without executing the mutation and q was reached by executing the mutation. Alternatively, product state equivalence may also be checked instead. However, this choice has been made by comparing the performance gain resulting from pruning to the performance loss resulting from equivalence checks. While checking equivalence is computationally expensive, actions can be executed efficiently for a product state that is not affected by the mutation. Consequently, pruning is not necessary in a state if the mutation did not take effect. This is reflected by the decision to identify states to be inequivalent in the discussed situation.

It shall now be shown that $\equiv_{approx}$ is a stronger condition than $\equiv_{prod}$ and thus is a valid approximation of $\equiv_{prod}$. A stronger condition is a valid approximation because less traces will be pruned if it is used as pruning criterion. This leads to the exploration of redundant traces, which does not introduce incorrect behaviour. The conformance-checking algorithm will still produce the same result if this optimisation is applied. Although the exploration of redundant paths contradicts the strategy presented before, this approximation shall be used because the performance increase resulting from its usage will in general be significantly higher than the performance loss caused by the exploration of redundant paths. Given two conditions c and c', c is stronger than c' if $c \rightarrow c'$ is a tautology.

It shall be shown that: $((\kappa, \mu), e) \equiv_{approx} ((\kappa', \mu'), e') \rightarrow (\kappa, \mu) \equiv_{prod} (\kappa', \mu')$

$$\begin{aligned} ((\kappa, \mu), e) \equiv_{approx} ((\kappa', \mu'), e') = & (R(\mu) = R(\mu') \wedge e = True \wedge e' = True) \vee \\ & (\bot \wedge e = True \wedge e' = False) \vee \\ & ((\kappa, \mu) \equiv_{prod} (\kappa', \mu') \wedge e = False \wedge R(\mu) = R(\mu')) \vee \\ & (\bot \wedge e = False \wedge R(\mu) \neq R(\mu')) \end{aligned}$$

$$
\begin{aligned}
&= (R(\mu) = R(\mu') \wedge e = True \wedge e' = True) \vee \\
&\quad ((\kappa, \mu) \equiv_{prod} (\kappa', \mu') \wedge e = False \wedge R(\mu) = R(\mu')) \\
&\quad \rightarrow (\kappa, \mu) \equiv_{prod} (\kappa', \mu')
\end{aligned}
$$

The implication holds if (a) $R(\mu) = R(\mu') \wedge e = True \wedge e' = True$

$$\rightarrow (\kappa, \mu) \equiv_{prod} (\kappa', \mu')$$

and (b) $(\kappa, \mu) \equiv_{prod} (\kappa', \mu') \wedge e = False \wedge R(\mu) = R(\mu')$

$$\rightarrow (\kappa, \mu) \equiv_{prod} (\kappa', \mu')$$ hold

(a) holds because: $R(\mu) = R(\mu') \wedge e = True \wedge e' = True$

$$\rightarrow R(\mu) = R(\mu') \rightarrow (\kappa, \mu) \equiv_{prod} (\kappa', \mu') \ldots$$ shown above

(b) holds because: $(\kappa, \mu) \equiv_{prod} (\kappa', \mu') \wedge e = False \wedge R(\mu) = R(\mu')$

$$\rightarrow (\kappa, \mu) \equiv_{prod} (\kappa', \mu') \ldots$$ consequent is part of antecedent

□

4.7 Reducing the Number of Non-conformance Checks

The observation that implementation and specification state are equivalent until the mutation takes effect can also help to reduce the number of checks of the non-conformance condition. Intuitively, the implementation is likely to show conforming behaviour in the next step as long as both implementation and specification are in the same state. This observation shall now be investigated more thoroughly, which leads to the distinction of four different cases.

1. If the mutated action has not been executed yet and the state update of an action or the guard of an input action has been mutated then it is not necessary to execute non-conformance checks.
2. If the mutated action has not been executed yet and the guard of an internal action has been mutated then it suffices to perform the non-conformance check for the quiescence observation.
3. If the mutated action has not been executed yet and the guard of an output action has been mutated then it suffices to perform the non-conformance check for the quiescence observation and the mutated output action.
4. If the init block has been mutated or the mutated action has already been executed then it is necessary to perform the non-conformance check for all observations.

In the following, each of the four cases shall be discussed and it shall be shown that optimisations based on the first three cases are valid. Hence, it is necessary to show that the application of the optimisations does not lead to situations, in which non-conformance conditions are not detected. In other words, it will be shown that the reduction of non-conformance checks does not lead to incorrect conformance checking verdicts. This is done by showing that all ignored non-conformance conditions would be unsatisfiable anyway.

For this purpose, it shall now be assumed that the conditions mentioned in the first case hold. More concretely, it is assumed that the implementation state κ is the same as the specification state μ and that the mutation does not affect the guard of an internal or an output action. It is important to note that the states of the implementation and the specification are syntactically equivalent which is a stricter requirement than semantical equivalence defined by $\equiv_{com}$.

Given a product state (κ, μ) with $\kappa = \mu$, it shall be shown that the non-conformance condition is unsatisfiable:

$$\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S \wedge \varphi_S[\pi_S]\right) \overset{!}{\leftrightarrow} \bot$$

for all $\lambda \in \Lambda_U \cup \{\delta\}$ and $(\chi, \pi) \in \mu$

where $\exists \rho_P : (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\}$

and $\exists \rho_S : (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\}$

$$\overset{\kappa=\mu}{\leftrightarrow} \left(\bigvee_{(\gamma,\pi)\in\kappa} \gamma \wedge \varphi_P[\pi]\right) \wedge \neg \left(\bigvee_{(\gamma,\pi)\in\kappa} \gamma \wedge \varphi_S[\pi]\right) \wedge \chi$$

$$\overset{\varphi_P=\varphi_S=\varphi}{\leftrightarrow} \left(\bigvee_{(\gamma,\pi)\in\kappa} \gamma \wedge \varphi[\pi]\right) \wedge \neg \left(\bigvee_{(\gamma,\pi)\in\kappa} \gamma \wedge \varphi[\pi]\right) \wedge \chi$$

$$\overset{A\wedge\neg A=\bot}{\leftrightarrow} \bot \wedge \chi \leftrightarrow \bot$$

□

The second rewriting operation requires that $\varphi_P = \varphi_S$ for all $\lambda \in \Lambda_U \cup \{\delta\}$. This is obviously fulfilled for all $\lambda \in \Lambda_U$ given the assumption that the mutation does not affect the guard of an output action. Recall that the guard Δ of the quiescence observation is defined as

$$\Delta = \bigwedge \left\{ \neg \overline{\exists}_{\mathsf{para}(\lambda)} \psi \;\middle|\; \exists \rho : (\lambda, \psi, \rho) \in \rightarrow \text{ with } \lambda \in \Lambda_U \cup \{\tau\} \right\},$$

thus it depends on the guards of internal and outputs actions. Since the assumption placed above also states that the mutation must not affect the guards of internal actions, the quiescence conditions of specification and implementation are identical. Hence, it is not possible to miss conformance violations by skipping checks for these if the assumptions listed above are fulfilled.

However, if the conditions of the second case are fulfilled then $\varphi_P = \varphi_S$ does not hold for all $\lambda \in \Lambda_U \cup \{\delta\}$ because the condition for the quiescence observation depends on the guards of internal actions. Consequently, it is necessary to perform non-conformance checks for the quiescence observation if the guard of an internal action is mutated. If the guard of an output action is mutated then $\varphi_P \neq \varphi_S$ for one $\lambda \in \Lambda_U$ and for δ because the quiescence condition also depends on the guards of output actions. Hence, it necessary to perform non-conformance checks for the mutated action and for the quiescence observation. The other output actions need not be checked though as the guards of the unmutated outputs are equivalent in implementation and specification.

4.8 Calculation of Reachable Actions

The following optimisation is essentially an extension of the optimisation presented in Section 4.7. More concretely, it checks whether the assumptions of the first discussed case are fulfilled until the maximum search depth is hit. In other words, it checks whether the mutated action may be executed before hitting the depth bound. If it is not executable and additionally the mutation does not affect the guard of an internal or an output action, then the search can be stopped. This is possible because non-conformance would not be detected anyway.

For this purpose, the notion of reachable actions shall be introduced. An action a is reachable from some state q if it can be executed in q, or if a state q' can be reached by executing an action different from a, from which a is reachable. Reachability of an action a can be further refined to include an upper bound on the maximum number of intermediate action executions. An action a is reachable within s

steps from a state q if $s > 0$ and a is executable or if a state q' can be reached by executing an action different from a and a is reachable within $s - 1$ steps from q'.

If the sets of reachable actions are known for the specification, the optimisation presented in Section 4.7 can be further extended. As noted above, this information can be utilised if the assumptions of the first discussed case are fulfilled. If additionally to those assumptions, it holds that the mutated action can not be performed in subsequent steps then it is not necessary to continue the search for conformance violations.

Since the conformance check is performed for all traces of the specification, the set of actions reachable from any given state can be determined in a precomputation step based on the symbolic execution graph. Considering internal actions, this information is sufficient as well, because of the assumption that the mutation does not affect the guard of an internal action. In this case, it is not possible for the implementation to reach an internal action which is not reachable for the specification.

Hence, another data structure $reachableAct$ shall be introduced, which maps pairs consisting of graph node identifiers and integral depth values to sets of action names. It shall be possible to retrieve the set of reachable actions for a graph node at a given depth d_{curr} smaller than the maximum search depth d. Stated differently, it shall be possible to retrieve the actions reachable from a graph node within a given number of steps s, where the relation between s and d is given by $s = d - d_{curr}$.

Pairs consisting of a graph node identifier id and depth information d_r shall be used as keys for key-value pairs stored in $reachableAct$. The corresponding values are sets of reachable action names r. This enables querying of reachable actions r from a node with identifier id within d_r steps.

The data structure $reachableAct$ is assumed to be globally accessible in the following and is created by executing Algorithm 5. The quiescence observation will, like in Algorithm 3, be treated identically to actions. In order to calculate the correct depth information, the symbolic execution tree needs to be fully explored up to depth d, thus making it necessary to unfold the loops in the symbolic execution graph. Nevertheless, the symbolic state equivalence classes can still be used to speed up the computation by utilising the fact that the sets of reachable actions are the same for equivalent states. Therefore, it is sufficient to calculate the successors only once per equivalence class. As a result, queries for reachable actions have to be performed for the representative corresponding to a symbolic graph node. The algorithm makes use of the auxiliary function defined below.

1: **function** REACHABLEINTERNALS($nodeID$)
2: **return** REACHABLEINTERNALSREC($\{\}$,$\{nodeID\}$, $\{\}$)
3: **end function**
4: **function** REACHABLEINTERNALSREC($nodes$,$newNodes$,$internals$)
5: $nextNodes \leftarrow \{id' \mid \exists q, q' : id \in newNodes \wedge ((q, rep(id)), label, (q', id')) \in graphEdges \wedge id' \notin nodes \wedge label$ is an internal action name$\}$
6: $internals \leftarrow internals \cup \{label \mid \exists q, q', id' : id \in newNodes \wedge ((q, rep(id)), label, (q', id')) \in graphEdges \wedge label$ is an internal action name$\}$
7: **if** $nextNodes = \{\}$ **then**
8: **return** $internals$
9: **else**
10: **return** REACHABLEINTERNALSREC($nodes \cup newNodes$,$nextNodes$,$internals$)
11: **end if**
12: **end function**

The function $reachableInternals$ basically explores all nodes reachable from a given start node, denoted by $nodeID$, by executing internal actions. For this purpose, it calls a recursive function which explores all immediately reachable nodes from the currently considered set of nodes $newNodes$, which is initialised with the singleton set $\{nodeID\}$. In Line 5, the successor nodes of $newNodes$ are computed and filtered such that no node is explored twice and all nodes are reached solely by internal actions.

Algorithm 5 Calculation of reachable actions.

1: **procedure** CALCREACHABLEACTINIT
2: $reachableAct \leftarrow emptyMap$
3: $(q, startID) \leftarrow startNode$
4: CALCREACHABLEACTREC($startID$, d)
5: **end procedure**
6: **procedure** CALCREACHABLEACTREC($nodeID$, $remDepth$)
7: **if** $remDepth = -1$ **then**
8: $put(reachableAct, (rep(nodeID), remDepth)$,REACHABLEINTERNALS($nodeID$))
9: **else if** $get(reachableAct, (rep(nodeID), remDepth))$ is not defined **then**
10: $nextTrans \leftarrow \{t \mid t \in graphEdges \wedge \exists q, q', l, id' : t = ((q, rep(nodeID)), l, (q', id'))\}$
11: $currReachable \leftarrow \{\}$
12: **for all** $((q, rID), label, (q', id')) \in nextTrans$ **do**
13: $nextDepth \leftarrow \begin{cases} remDepth & \text{if } label \text{ is an internal action name} \\ remDepth - 1 & \text{otherwise} \end{cases}$
14: **if** $get(reachableAct, (rep(id'), nextDepth))$ is defined **then**
15: $currReachable \leftarrow currReachable \cup get(reachableAct, (rep(id'), nextDepth)) \cup \{label\}$
16: **else**
17: CALCREACHABLEACTREC(id',$nextDepth$)
18: $currReachable \leftarrow currReachable \cup get(reachableAct, (rep(id'), nextDepth)) \cup \{label\}$
19: **end if**
20: **end for**
21: $put(reachableAct, (rep(nodeID), remDepth), currReachable)$
22: **end if**
23: **end procedure**

Line 6 is formed similarly, but determines the corresponding internal action labels. The function terminates if the set of nodes scheduled for further exploration is empty (Lines 7 und 8). Otherwise, the search is continued recursively.

Like the auxiliary function $reachableInternals$, Algorithm 5 consist of an initialisation and a recursive part. In the initialisation, the data structure $reachableAct$ is set to be empty, the exploration depth is set to the maximum depth d, and the node to be explored next is set to be the start node of the symbolic execution graph (Lines 1 to 5).

In the recursive part of the algorithm, a variant of depth-first exploration is performed. If the remaining depth $remDepth$ is -1, then only internal actions reachable from the current node, denoted by $nodeID$, need to be determined (Lines 7 and 8). Otherwise, it is checked whether a node in the same equivalence class as $nodeID$ has already been explored (Line 9). If this is the case, it is not necessary to continue the search along the current path. Hence, the exploration strategy is not purely depth-first.

In order to explore a node n, all actions executable in n are determined (Line 10) and the set of reachable actions $currReachable$ is initially set to be empty for n (Line 11). Afterwards, each action is processed (Lines 12 to 20) whereby:

- it is either determined that a state equivalent to the post-state of the action has already been explored (Line 14 and 15),
- or the post-state of the action is recursively explored (Lines 16 to 18).

Furthermore, $currReachable$ is incrementally extended by adding immediately reachable actions and actions reachable from post-states (Line 14 and 15 and Lines 16 to 18).

After processing all executable actions $nextTrans$, the set of all actions reachable from $nodeID$ within $remDepth$ steps is added to $reachableAct$.

Given the data structure $reachableAct$, it is possible to define a predicate $mutActionReachable$. This predicate takes a compound symbolic state associated with symbolic graph nodes as parameter and can be used to test if the mutated action may be executed before the maximum search depth is reached.

$$mutActionReachable : \mathcal{P}((\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \times ID) \to \{\top, \bot\}$$
$$mutActionReachable : \kappa \mapsto \exists((\varphi, \rho)_i, id) \in \kappa : mutAction \in get(reachableAct, (rep(id), d - i))$$

The predicate $mutActionReachable$ derives depth information from the index of a symbolic state and can be used in conjunction with other checks in the `if`-condition in Line 18 of Algorithm 2. If the predicate evaluates to $\bot$, then the `then`-branch of the `if`-statement should not be executed, which prevents further exploration following the current state. Hence, this optimisation also prunes the product graph implicitly.

4.9 Filtering of Implementation States

Compound symbolic states of the implementation may contain symbolic states which are irrelevant for the non-conformance check and thus may be discarded. A symbolic state s is irrelevant, if there exists no valuation which satisfies both the path condition of s and the path condition of a state of the specification. These states may exist for conforming implementations because **ioco** and thereby also **sioco** allows for implementation freedom for non-specified inputs [78]. This is achieved by testing only suspension traces of the specification. Hence, irrelevant states could also be referred to as underspecified states since the behaviour in those states is not specified.

Although such irrelevant states can only be reached by the implementation, they are visited during the conformance check because of the symbolic handling of data. Especially the "negated" execution of inputs, the application of $exec_{neg}$, may lead to irrelevant states. The cause for this is that first-order mutants show behaviour similar to that of the specification and $exec_{neg}$ negates the guard of an action.

This becomes apparent when considering an angelic completion for an unmutated action of a deterministic action systems which has not yet executed the mutation. In this case, the positive guard is added to the symbolic state of the specification and the negated guard is added to the symbolic state of the implementation. Note that there is only one symbolic state in the compound states prior to the angelic completion because the considered action systems are deterministic. Since the original and the negated guard cannot be satisfied at the same time, the state reached by angelic completion is actually irrelevant. However, through the restriction of angelic completion discussed in Section 4.4, this situation cannot occur actually. Nevertheless, it serves to demonstrate that $exec_{neg}$ may create irrelevant states.

Hence, the compound symbolic states computed through the application of $exec_{neg}$ shall be filtered such that all irrelevant states are discarded. Before giving the definition of a function for filtering compound symbolic states, the term irrelevant state shall be defined formally and it shall be shown that irrelevant states are indeed irrelevant.

Definition 4.3 (Irrelevant States).
Let (κ, μ) be a product state, such that κ is a compound symbolic state of the implementation and μ is a compound symbolic state of the specification. A state $(\varphi, \rho) \in \kappa$ is irrelevant if:

$$\nexists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \left(\bigvee_{(\psi,\pi)\in\mu} \psi \right)$$

The following proposition can be derived from the definition of irrelevant states.

Proposition 4.1.
Let (κ, μ) be a product state, such that κ is a compound symbolic state of the implementation and μ is a compound symbolic state of the specification, and let $(\varphi, \rho) \in \kappa$ be an irrelevant state. It holds that:

$$\forall(\psi, \pi) \in \mu : \varphi \wedge \psi \leftrightarrow \bot$$

It shall now be shown that irrelevant states may be ignored for the non-conformance check. This will be done by rewriting a non-conformance condition formed for a product state containing irrelevant states. The resulting formula shall not depend on irrelevant states.

Irrelevant States in Non-conformance Condition.

Let (κ, μ) be a product state containing irrelevant states.

The non-conformance condition for this state is given by:

$$\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \underbrace{\neg\left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S \wedge \varphi_S[\pi_S]\right)}_{\xi}$$

where $(\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup\{(\delta, \Delta_P, \mathsf{id})\}$
and $(\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup\{(\delta, \Delta_S, \mathsf{id})\}$
and $\exists\pi : (\chi, \pi) \in \mu$

$$\Leftrightarrow \left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa\setminus\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \vee \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right)\right) \wedge \chi \wedge \xi$$

where $\kappa_{irr} \subseteq \kappa \wedge \forall(\varphi, \rho) \in \kappa_{irr} : \nexists\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \bigvee_{(\psi,\pi)\in\mu} \psi$

and $\forall(\varphi, \rho) \in \kappa \setminus \kappa_{irr} : \exists\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \bigvee_{(\psi,\pi)\in\mu} \psi$

$$\Leftrightarrow \left(\left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa\setminus\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi\right) \vee \left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi\right)\right) \wedge \xi$$

$$\Leftrightarrow \left(\left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa\setminus\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi\right) \vee \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{irr}} \gamma_P \wedge \chi \ \wedge \varphi_P[\pi_P]\right)\right) \wedge \xi$$

since $\forall(\varphi, \rho) \in \kappa_{irr}, \forall(\psi, \pi) \in \mu : \varphi \wedge \psi \leftrightarrow \bot$ (Proposition 4.1) and $\exists\pi : (\chi, \pi) \in \mu$

$$\Leftrightarrow \left(\left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa\setminus\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi\right) \vee \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{irr}} \bot\right)\right) \wedge \xi$$

$$\Leftrightarrow \left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa\setminus\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi\right) \wedge \xi$$

$$\Leftrightarrow \left(\bigvee_{(\gamma_P,\pi_P)\in\kappa\setminus\kappa_{irr}} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \neg\left(\bigvee_{(\gamma_S,\pi_S)\in\mu} \gamma_S \wedge \varphi_S[\pi_S]\right)$$

□

Since it has been proven that the non-conformance condition does not depend on irrelevant states, it remains to be shown that an irrelevant state will not lead to states relevant for the conformance check.

Let (κ, μ) be a product state which contains an irrelevant state (γ, η). It shall be shown that the execution of actions in (γ, η) will lead to irrelevant states. Let $(\varphi', \rho') \in \kappa'$ and $(\psi', \pi') \in \mu'$ be states reached after executing a trace σ in (κ, μ), that is, $(\kappa, \mu) \xrightarrow{\sigma} (\kappa', \mu')$. They contain path conditions of the form

$$\varphi' = \varphi \wedge \bigwedge_{\lambda \in \sigma} g_{\lambda_P} \text{ and } \psi' = \psi \wedge \bigwedge_{\lambda \in \sigma} g_{\lambda_S}$$

for some $(\varphi, \rho) \in \kappa$ and $(\psi, \pi) \in \mu$, where g_{λ_S} and g_{λ_P} denote the guards of executed actions. The guards may be negated for the implementation.

Hence, κ' may contain states (γ', η') with path condition $\gamma' = \gamma \wedge \bigwedge_{\lambda \in \sigma} g_{\lambda_P}$ which shall be shown to be unsatisfiable. As assumed above, (γ, η) is irrelevant and thus

$$\gamma \wedge \left(\bigvee_{(\psi, \pi) \in \mu} \psi \right)$$

is unsatisfiable. This implies that

$$\gamma \wedge \bigwedge_{\lambda \in \sigma} g_{\lambda_P} \wedge \left(\bigvee_{(\psi, \pi) \in \mu} \psi \right) = \gamma' \wedge \left(\bigvee_{(\psi, \pi) \in \mu} \psi \right)$$

is unsatisfiable as well. It follows that

$$\gamma' \wedge \left(\bigvee_{(\psi, \pi) \in \mu} \psi \wedge \left(\bigwedge_{\lambda \in \sigma} g_{\lambda_S} \right) \right) = \gamma' \wedge \left(\bigvee_{(\psi', \pi') \in \mu'} \psi' \right)$$

is also unsatisfiable and that (γ', η') is irrelevant.

It can be concluded that irrelevant states lead to irrelevant successor states and thus can be filtered out. An intuitive argument for this fact is that adding further constraints to an unsatisfiable formula will yield another unsatisfiable formula. Adding constraints corresponds to the execution of actions.

Since it has been shown that irrelevant states can be ignored, it is possible to discard them as soon as they appear. However, as the detection of irrelevant states requires further satisfiability checks, this shall only be performed after the application of $exec_{neg}$. The reason for this decision is that it is likely that $exec_{neg}$ creates such states although $exec$ may create irrelevant state as well. Another auxiliary function filter shall be introduced for this purpose:

$$\mathit{filter} : \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{P}} \times \mathbb{N}_0) \times \mathfrak{F}(\widehat{\mathcal{I}}) \to \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{P}} \times \mathbb{N}_0)$$
$$\mathit{filter} : (\kappa, \psi) \mapsto \left\{ (\varphi, \rho)_i \;\middle|\; (\varphi, \rho)_i \in \kappa \wedge \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \psi \right\}$$

Let (κ, μ) be a product state: filter is intended to be used in Line 21 of Algorithm 2 as follows:

$$\mathit{filter}(exec_{neg}(\kappa, \gamma, \gamma_\tau), \psi) \text{ where } \psi = \bigvee_{(\varphi, \rho) \in \mu} \varphi \text{ is the path condition of the specification}$$

Like other optimisations presented, this optimisation helps to reduce the number of satisfiability checks necessary to execute actions. Assuming that this optimisation would not be applied, actions might be executed in irrelevant states, thus requiring satisfiability checks for path conditions of post-states which are also irrelevant. Moreover, its application leads to less complex path condition formulas of product states. This simplifies non-conformance and equivalence checks and may thereby further enhance performance.

Note that removing symbolic implementation states from a product state (κ, μ) results in a product state (κ', μ'), such that $(\kappa, \mu) \equiv_{prod} (\kappa', \mu')$. This can be shown by considering the interpretations of product states. Irrelevant states have been characterised as states $(\varphi, \rho) \in \kappa$ such that $\varphi \wedge \left(\bigvee_{(\psi,\pi)\in\mu} \psi\right)$ is unsatisfiable. Hence, a valuation may either satisfy an irrelevant state or some states of the specification. Recall the definition of product state interpretations with respect to a valuation υ and let υ be a valuation such that $\upsilon \models \varphi$, where φ is the path condition of an irrelevant state (φ, ρ):

$$\begin{aligned}
[\![(\kappa,\mu)]\!]_\upsilon &= \bigcup_{(\varphi,\rho)\in\kappa} \{\upsilon_{\mathsf{eval}} \circ \rho \mid \upsilon \models \varphi\} \times \bigcup_{(\psi,\pi)\in\mu} \{\upsilon_{\mathsf{eval}} \circ \pi \mid \upsilon \models \psi\} \\
&= \bigcup_{(\varphi,\rho)\in\kappa} \{\upsilon_{\mathsf{eval}} \circ \rho \mid \upsilon \models \varphi\} \times \emptyset \text{ because } \forall (\psi,\pi) \in \mu : \upsilon \not\models \psi \\
&= \emptyset
\end{aligned}$$

It can be concluded that all interpretations with respect to valuations, which satisfy path conditions of irrelevant states, will be empty. Consequently irrelevant states do not have any influence on the set of all interpretations $[\![(\kappa, \mu)]\!] = \bigcup_{\upsilon\in\mathfrak{U}^{\widehat{\mathcal{I}}}} [\![(\kappa, \mu)]\!]_\upsilon$ and thereby they do not affect $\equiv_{prod}$ as well.

4.10 Checking if Input Guard Weakened

The last optimisation, which shall be presented in this section is only applicable for deterministic action systems. Nevertheless, it can significantly decrease the **sioco** checking run-time. Intuitively, a mutant conforms to the specification, from which it was derived, if it does not change the behaviour of the specification, but rather extends it. This is the case in the context of **ioco** because of the aforementioned freedom of implementation for non-specified inputs [78]. Considering first-order mutants, a mutant will conform to the specification, if the guard of an input action is mutated in a way such that it accepts all inputs also accepted by the specification without performing an angelic completion.

A simple **sioco**-conformance condition based on the observations given above shall now be derived. Hence, implementations are assumed to be first-order mutants of the specification such that the mutation affects the guard of an input action. For this purpose, the interpretation of action systems as IOLTSs shall be considered. Let $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ be an action system and let $[\![\mathcal{AS}]\!] = \langle \mathfrak{U}^{\mathcal{V}}, \mathsf{eval} \circ \iota, \Sigma_I, \Sigma_U, \rightarrow_{LTS} \rangle$ be its interpretation. For a state $q \in \mathfrak{U}^{\mathcal{V}}$, the set of all inputs accepted in state q can be defined by:

$$I_{acc}(q) =_{def} \{(\lambda, p) \mid \exists q' : (q, (\lambda, p), q') \in \rightarrow_{LTS} \wedge (\lambda, p) \in \Sigma_I\}$$

Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$ be a specification and let $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$ be an implementation that adheres to the assumptions given above, thus it is a first-order mutant of the specification $\mathcal{AS}_S$. Their interpretations are given by $[\![\mathcal{AS}_S]\!] = \langle \mathfrak{U}^{\mathcal{V}_S}, \mathsf{eval} \circ \iota_S, \Sigma_I, \Sigma_U, \rightarrow_{LTS_S} \rangle$ and $[\![\mathcal{AS}_P]\!] = \langle \mathfrak{U}^{\mathcal{V}_P}, \mathsf{eval} \circ \iota_P, \Sigma_I, \Sigma_U, \rightarrow_{LTS_P} \rangle$ respectively. The synchronous product graph of $[\![\mathcal{AS}_S]\!]$ and $[\![\mathcal{AS}_P]\!]$ defined by the rules given by Weiglhofer and Wotawa [83] is an IOLTS $SP = [\![\mathcal{AS}_P]\!] \times_{ioco} [\![\mathcal{AS}_S]\!]$, where $SP = \langle Q_{SP}, s_{0_{SP}}, \Sigma_I, \Sigma_U, \rightarrow_{SP} \rangle$, $Q_{SP} = \mathfrak{U}^{\mathcal{V}_P} \times \mathfrak{U}^{\mathcal{V}_S} \cup \{fail, pass\}$ and $s_{0_{SP}} = (\mathsf{eval} \circ \iota_P, \mathsf{eval} \circ \iota_S)$. It is now possible to give a non-symbolic conformance-condition:

$[\![\mathcal{AS}_P]\!]$ **ioco** $[\![\mathcal{AS}_S]\!]$ and thus also $\mathcal{AS}_P$ **sioco** $\mathcal{AS}_S$
if: $\forall (q_P, q_S) \in Q_{SP} : I_{acc}(q_S) \subseteq I_{acc}(q_P)$ and the assumptions listed above are fulfilled

Intuitively, this is fulfilled if the mutated guard is weaker and thus accepts more inputs than the guard of the specification. Stated conversely, it is fulfilled if the guard of the specification is stronger than the mutated guard. As noted before, a stronger condition implies a weaker condition, so it could be checked if an implication between the guards is a tautology. Hence, the subset-relation between accepted inputs could be modelled via implication.

However, the non-symbolic conformance-condition contains another constraint. The quantification $\forall(q_P, q_S) \in Q_{SP}$ requires that the states may not be freely chosen, but must be contained in Q_{SP}. As the mutation only affects a guard, the states q_P and q_S of a product state (q_P, q_S) are equal.

To be able to formulate this constraint in the **sioco** context a bijection $mutv : \mathcal{V}_S \to \mathcal{V}_P$ shall be introduced. It maps specification variables to their corresponding implementation variables. Such a function exists since the implementation is a first-order mutant.

Proposition 4.2 (Weakened Guard Conformance Condition).
Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \to_S \rangle$ be a specification and let $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \to_P \rangle$ be an implementation such that $\mathcal{AS}_P$ is a first-order mutant of $\mathcal{AS}_S$ and the mutation affects the guard of an input action $\lambda_{mut} \in \Lambda_I$. Assuming additionally that $\bigwedge_{v \in \mathcal{V}_S} v = mutv(v)$ is fulfilled, it holds that:

$$\forall d \in \mathbb{N}_0 : \ \mathcal{AS}_P \ \mathbf{sioco}_d \ \mathcal{AS}_S \textit{ if } (\varphi_S \to \varphi_P) \Leftrightarrow \top$$
$$\textit{where } \exists \pi_S : (\lambda_{mut}, \varphi_S, \pi_S) \in \to_S \textit{ and } \exists \pi_P : (\lambda_{mut}, \varphi_P, \pi_P) \in \to_P$$

It shall now be shown that this proposition holds. In other words, it shall be shown that weakening the guard of an input action produces a conforming mutant. The proof will be performed in two steps. In the first step, it will be shown that the execution of actions leads to product states containing exactly one relevant state and that $\bigwedge_{v \in \mathcal{V}_S} v = mutv(v)$ holds for these relevant states. The second step is based on the first step and will prove that the non-conformance condition is unsatisfiable for all product states in the deterministic symbolic synchronous product graph $\mathcal{AS}_P \times_{sioco_{det}} \mathcal{AS}_S(d)$.

Step 1. It is possible to distinguish two cases for this proof, the execution of input actions and the execution of output actions. The execution of input action actions shall be investigated first.

Therefor, a proof by induction shall be performed for the statement that the execution of input actions in a product state (κ, μ), with $\mu = \{(\varphi, \rho)\}$ and κ containing exactly one relevant state (ψ, ρ) will yield a product state (κ', μ'), with $\mu' = \{(\varphi', \rho')\}$ and κ' containing exactly one relevant state (ψ', ρ'). Note that the symbolic state vectors ρ and ρ' are the same for both specification and implementation.

For the induction base, a product state (κ, μ) reached by executing a sequence of output actions shall be considered. Let (κ, μ) be a product state, such that $q_{init} \stackrel{\sigma}{\Rightarrow} (\kappa, \mu)$ for some $\sigma \in (\Lambda_U \cup \{\delta\})^*$ and let (κ', μ') be a product state, such that $(\kappa, \mu) \stackrel{\lambda_i}{\Rightarrow} (\kappa', \mu')$ for an input $\lambda_i \in \Lambda_I$. Since σ contains only non-mutated output actions $exec_{neg}$ is not applied for any action in σ. Hence, it holds that $\kappa = \mu = \{(\varphi, \rho)\}$. The execution of an input action λ_i leads to the states

$$\kappa' = \{(\varphi \wedge \psi_P[\rho], \rho'), (\varphi \wedge \neg\psi_P[\rho], \rho)\} \text{ and}$$
$$\mu = \{(\varphi \wedge \psi_S[\rho], \rho')\}, \text{ where } (\lambda_i, \psi_P, \pi) \in \to_P \text{ and } (\lambda_i, \psi_S, \pi) \in \to_S .$$

The state $(\varphi \wedge \psi_P[\rho], \rho')$ and $(\varphi \wedge \psi_S[\rho], \rho')$ have the same symbolic state vector because the mutation does not affect the state update π. It shall now be shown that $(\varphi \wedge \neg\psi_P[\rho], \rho)$ is an irrelevant state.

$\varphi \wedge \neg\psi_P[\rho] \wedge \varphi \wedge \psi_S[\rho] \stackrel{!}{\Leftrightarrow} \bot$
$\Leftrightarrow \varphi \wedge (\psi_S[\rho] \wedge \neg\psi_P[\rho])$
Case 1: $\lambda_i \neq \lambda_{mut} \Rightarrow \psi_P = \psi_S$
$\Leftrightarrow \varphi \wedge (\psi_S[\rho] \wedge \neg\psi_S[\rho]) \Leftrightarrow \varphi \wedge \bot \Leftrightarrow \bot$
Case 2: $\lambda_i = \lambda_{mut} \Rightarrow (\psi_S[\rho] \to \psi_P[\rho]) \leftrightarrow \top \Rightarrow \neg(\psi_S[\rho] \to \psi_P[\rho]) \leftrightarrow \bot \Leftrightarrow (\psi_S[\rho] \wedge \neg\psi_P[\rho]) \leftrightarrow \bot$
$\varphi \wedge (\psi_S[\rho] \wedge \neg\psi_P[\rho]) \Rightarrow \varphi \wedge \bot \Leftrightarrow \bot$

The base step fulfils the statement which shall be proven because $(\varphi \wedge \neg\psi_P[\rho], \rho)$ has been shown to be an irrelevant state. Furthermore, the only relevant state $(\varphi \wedge \psi_P[\rho], \rho')$ in κ' contains the same symbolic state vector ρ' as the specification state $(\varphi \wedge \psi_S[\rho], \rho')$.

For the induction step, a product state (κ, μ) reached after an arbitrary sequence of actions shall be considered. According to the induction hypothesis $\mu = \{(\varphi_S, \rho)\}$ and $\kappa = \{(\varphi_P, \rho)\} \cup \kappa_{irr}$, such that (φ_P, ρ) is a relevant state and κ_{irr} contains only irrelevant states. The execution of an input action λ_i leads to the product state (κ', μ'), such that

$$\kappa' = \{(\varphi_P \wedge \psi_P[\rho], \rho'), (\varphi_P \wedge \neg\psi_P[\rho], \rho)\} \cup \kappa'_{irr} \text{ and}$$
$$\mu = \{(\varphi_S \wedge \psi_S[\rho], \rho')\}, \text{ where } (\lambda_i, \psi_P, \pi) \in \rightarrow_P \text{ and } (\lambda_i, \psi_S, \pi) \in \rightarrow_S .$$

Since it has been shown in Section 4.9 that the execution of actions in irrelevant states will lead to irrelevant post-states, κ'_{irr} does not contain relevant states. It has to be shown that $(\varphi_P \wedge \neg\psi_P[\rho], \rho)$ is an irrelevant state.

$\varphi_P \wedge \neg\psi_P[\rho] \wedge \varphi_S \wedge \psi_S[\rho] \overset{!}{\Leftrightarrow} \bot$
$\Leftrightarrow (\varphi_P \wedge \varphi_S) \wedge (\psi_S[\rho] \wedge \neg\psi_P[\rho])$
Case 1: $\lambda_i \neq \lambda_{mut} \Rightarrow \psi_P = \psi_S$
$\Leftrightarrow (\varphi_P \wedge \varphi_S) \wedge (\psi_S[\rho] \wedge \neg\psi_S[\rho]) \Leftrightarrow (\varphi_P \wedge \varphi_S) \wedge \bot \Leftrightarrow \bot$
Case 2: $\lambda_i = \lambda_{mut} \Rightarrow (\psi_S[\rho] \rightarrow \psi_P[\rho]) \leftrightarrow \top \Rightarrow \neg(\psi_S[\rho] \rightarrow \psi_P[\rho]) \leftrightarrow \bot \Leftrightarrow (\psi_S[\rho] \wedge \neg\psi_P[\rho]) \leftrightarrow \bot$
$(\varphi_P \wedge \varphi_S) \wedge (\psi_S[\rho] \wedge \neg\psi_P[\rho]) \Rightarrow (\varphi_P \wedge \varphi_S) \wedge \bot \Leftrightarrow \bot$

Hence, the statement holds for the induction step as well and it follows that the statement is true. Since $exec_{neg}$ is not applied for output actions, the execution of outputs in a product state containing one relevant state will lead to post-states containing at most one relevant state. As the path condition of this state is required to be satisfiable, it must contain at least one relevant state. It follows that the execution of arbitrary actions in arbitrary states leads to post-states with exactly one relevant state.

The assumption $\bigwedge_{v \in \mathcal{V}_S} v = mutv(v)$ holds for all relevant states, because the mutation does not affect the state update mappings. It may be violated after the application of $exec_{neg}$ but it has been shown that applying $exec_{neg}$ leads to irrelevant states.

Step 2. It shall now be proven that the non-conformance condition is unsatisfiable under the given assumptions. Let (κ, μ) be product state, where $\mu = \{(\varphi_S, \rho)\}$ and $\kappa = \{(\varphi_P, \rho)\} \cup \kappa_{irr}$, such that κ_{irr} contains only irrelevant states and (φ_P, ρ) is a relevant state. Hence, the product state adheres to the conditions which have been shown to hold.

Unsatisfiability of Non-conformance Condition for Weakened Guards.

The non-conformance condition has the form:

$$\left(\bigvee_{(\gamma_P, \pi_P) \in \kappa} \gamma_P \wedge \psi[\pi_P] \right) \wedge \varphi_S \wedge \neg (\varphi_S \wedge \psi[\rho])$$

where $\lambda \in \Lambda_U \cup \{\delta\}$
and $\exists \eta_P : (\lambda, \psi_P, \eta_P) \in \rightarrow_P \cup \{(\delta, \Delta_P, \mathsf{id})\}$
and $\exists \eta_S : (\lambda, \psi_S, \eta_S) \in \rightarrow_S \cup \{(\delta, \Delta_S, \mathsf{id})\}$
and $\psi_P = \psi_S = \psi$ because an input is mutated

$$
\begin{aligned}
&\Leftrightarrow \quad \left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{irr}} \gamma_P \wedge \psi[\pi_P]\right) \vee (\varphi_P \wedge \psi[\rho])\right) \wedge \varphi_S \wedge \neg(\varphi_S \wedge \psi[\rho]) \\
&\Leftrightarrow \quad \left(\left(\bigvee_{(\gamma_P,\pi_P)\in\kappa_{irr}} \gamma_P \wedge \varphi_S \wedge \psi[\pi_P]\right) \vee (\varphi_P \wedge \psi[\rho] \wedge \varphi_S)\right) \wedge \neg(\varphi_S \wedge \psi[\rho]) \\
&\overset{irrelevant}{\Leftrightarrow} (\bot \vee (\varphi_P \wedge \psi[\rho] \wedge \varphi_S)) \wedge \neg(\varphi_S \wedge \psi[\rho]) \\
&\Leftrightarrow \quad (\varphi_P \wedge \psi[\rho]) \wedge (\varphi_S \wedge \neg(\varphi_S \wedge \psi[\rho])) \\
&\Leftrightarrow \quad (\varphi_P \wedge \psi[\rho]) \wedge (\varphi_S \wedge (\neg\varphi_S \vee \neg\psi[\rho])) \\
&\Leftrightarrow \quad (\varphi_P \wedge \psi[\rho]) \wedge ((\varphi_S \wedge \neg\varphi_S) \vee (\varphi_S \wedge \neg\psi[\rho])) \\
&\Leftrightarrow \quad (\varphi_P \wedge \psi[\rho]) \wedge (\varphi_S \wedge \neg\psi[\rho]) \Leftrightarrow \varphi_P \wedge \varphi_S \wedge (\psi[\rho] \wedge \neg\psi[\rho]) \Leftrightarrow \varphi_P \wedge \varphi_S \wedge \bot \Leftrightarrow \bot
\end{aligned}
$$

□

It follows that a simplified conformance check can be based on the observation that a mutation which solely weakens the guard of an input action of a deterministic specification produces a conforming mutant. Such a conformance check is given as a snippet of pseudo code in Algorithm 6. In contrast to Algorithm 2, which checks for non-conformance, Algorithm 6 rather checks for conformance. Furthermore, this algorithm can only be applied in conjunction with another conformance check because: (1) it requires strict preconditions and (2) if it is not successful, it cannot draw any meaningful conclusion. Stated differently, if all preconditions of Algorithm 6 are met and it detects that the guard of the mutated input action is not weakened, it cannot decide whether the considered mutant conforms to the specification. Such a decision requires further checks, for instance through the application of Algorithm 2.

Although the applicability of Algorithm 6 is limited, it introduces a significant speed-up in those cases where it can be applied. The performance gain results from the fact that generally, the product graph needs to be explored up to the maximum depth for conforming mutants. Furthermore, if the check detects that the guard is not weakened, it does not result in a large computational overhead as only one additional satisfiability check is needed. The check given in Algorithm 6 shall therefore be added to the `siocoCheckInit`-function defined in Algorithm 2.

Algorithm 6 first determines whether the considered action systems are indeed deterministic by checking whether they contain internal actions τ (Line 1). Afterwards, the other assumptions are verified to hold. If $\neg intOrOutMut$ is true (Line 1), then the mutation affects an input and if $\rho_S = \rho_P$ (Line 4), then the mutation affects the guard. The formula $\left(\bigwedge_{v\in\mathcal{V}_S} v = mutv(v)\right) \rightarrow (\varphi_S \rightarrow \varphi_P)$ is checked rather than $\varphi_S \rightarrow \varphi_P$ to account for the assumption that the states of implementation and specification are equal. If it is a tautology, the action systems are conforming (Lines 4 to 6).

Algorithm 6 Check if guard of mutated input action is weakened

1: **if** $|\{(\tau,\varphi,\psi) \mid (\tau,\varphi,\psi) \in \rightarrow_S\}| = 0 \wedge \neg intOrOutMut$ **then**
2: $(\lambda,\varphi_S,\rho_S) \leftarrow t_S$ such that $t_S \in \rightarrow_{\delta_S}$ and $actName(t_S) = mutAction$
3: $(\lambda,\varphi_P,\rho_P) \leftarrow t_P$ such that $t_P \in \rightarrow_{\delta_P}$ and $actName(t_P) = mutAction$
4: **if** $\rho_S = \rho_P \wedge \left(\bigwedge_{v\in\mathcal{V}_S} v = mutv(v)\right) \rightarrow (\varphi_S \rightarrow \varphi_P)$ is a tautology **then**
5: **return** *conforming*
6: **end if**
7: **else**
8: ... perform non-conformance check ...
9: **end if**

5 Application of the Conformance Check

The **sioco** conformance check discussed in the last two chapters may be used for both model-based mutation testing and model-checking. Hence, an overview of both intended application areas shall be given in this chapter. At first, the general structure of the model-based testing process proposed by this thesis will be described. However, two steps involved in the process will be discussed in more detail. The chapter will be concluded by a discussion of the applicability of the conformance check to model-checking.

5.1 The Model-Based Testing Process

The model-based testing process actually consists of two subprocesses, the test case generation process and the actual testing of applications using the derived test cases. In the first phase, several first-order mutants of a specification are generated. These mutants are then checked for conformance against the specification using the algorithm presented in Section 3.2. Test cases are derived from the results of the conformance check. These test cases and the original specification form the basis of the second stage, the test execution stage.

Since the type and effectiveness of test cases is governed by the mutation step performed in the first phase, this step will be investigated more closely. While other steps such as the mutation analysis, precomputation and the **sioco** check have already been discussed in great detail, test case execution has not been discussed so far. Hence, the actual execution of test cases shall also be examined narrowly, as it requires symbolic execution as well and thus is a complex task.

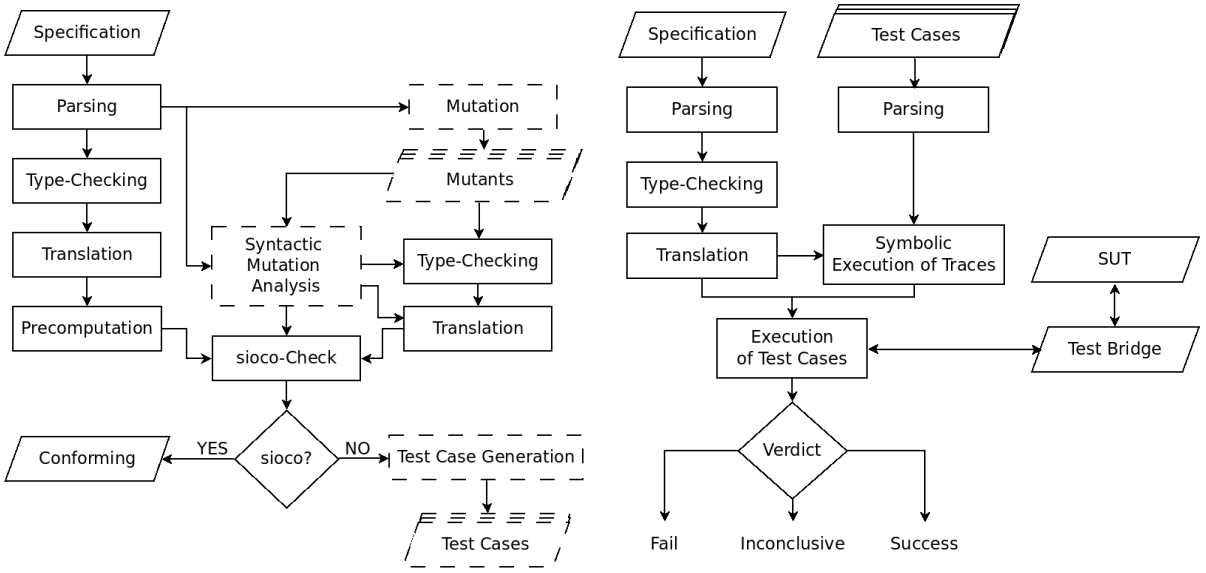
5.1.1 Test Case Generation Phase

The goal of the test case generation phase is to generate a set of abstract symbolic test cases based on a given specification and mutated versions of this specification. A test case shall produce a *fail*-verdict if it is executed on an SUT which implements the mutated specification corresponding to the test case. Hence, witnesses of non-conformance between mutants and specification shall serve as the basis for testing, as they encode conditions in which non-conformance of implementations may be observed. More specifically, in this thesis witnesses of non-conformance are actually considered symbolic test cases, which are processed in the test case execution phase. The process used to generate such test cases can be further subdivided into consecutively performed steps, which apply transformations on the specification and the results of preceding steps. A description of each of the steps and artefacts involved in the test case generation process will be given in the following, whereby their relation to each other will be highlighted. The process is also depicted in Figure 5.1a.

Specification. The specification is an action system written in the language defined in Section 2.3.1. It models the intended behaviour of a software system.

Parsing. The parsing step transforms the specification given as text into an Abstract Syntax Tree (AST). If the specification contains syntax errors, those are printed to the console and the test case generation is aborted.

Mutation. In the mutation step, the specification given as AST is systematically altered using a configurable set of mutation operators. A mutation operator inserts errors belonging to a certain class of errors into the specification. As a result, a set of mutated copies of the specification is produced. These mutated copies are also called mutants and since each mutant contains exactly one error, they are referred to as first-order mutants. The mutants serve as implementations for the conformance check. Optionally, they may be written to disk for debugging and documentation purposes.



(a) The steps performed during **sioco** checking. Steps and artefacts specific to model-based mutation testing have dashed borders.

(b) The steps and artefacts involved in the test case execution.

Figure 5.1: The model-based testing process

Syntactic Mutation Analysis. The syntactic mutation analysis performs the steps given by Algorithm 4, which include for instance the determination of the mutated action of a mutant.

Type-Checking. In the type checking step, type information is propagated to all nodes in the AST and error checks are performed. The test case generation is aborted, if the specification contains errors. This step utilises the results of the syntactic mutation analysis by considering only the mutated actions of mutants, as data concerning unmutated actions can be retrieved from type checking results of the specification.

Translation. In this step, the specification and the mutants are transformed into their respective symbolic representations. More specifically, the translation component maps ASTs containing type information into objects defined by the Java-Application Programming Interface (API) of the SMT-solver Z3[1] to express terms and formulas. As in the type-checking phase, results from the syntactic mutation analysis may be used for first-order mutants, because it is not necessary to translate unmutated actions.

Precomputation. During precomputation, a symbolic execution graph of the specification is created and sets of reachable actions are computed following the approaches described in Chapter 4.

sioco-Check. In this step, each mutant is checked for **sioco**-conformance against the specification. This either produces the verdict that the mutant conforms to the specification or it produces a witness of non-conformance. A witness of non-conformance as discussed in Section 3.1.3 is a pair consisting of a trace leading to an unsafe product state and a satisfiable formula capturing the non-conformance condition for a single action.

Test Case Generation. Since the original action system specification is processed during test execution to derive test verdicts, the implementation of the test case generation is rather simple. The symbolic test cases created in this step are mere serialisations of the witnesses of non-conformance, which are not further modified.

[1] A discussion concerning the choice of SMT-solver and API is given in Chapter 6.

Type of Test Cases and Mutation

The following discussion will focus on the type of generated test cases and the mutation step which influences these test cases. It aims at answering the question, as to what aspects are tested. Aichernig closely investigated the model-based mutation testing approach in his habilitation thesis [2]. Consequently, the discussion of tested aspects will be based on this thesis.

Mutation. However, prior to this, a technical overview of the implemented mutation component shall be given. As noted above the mutation step syntactically alters the specification creating a set of mutants. Stated differently, the mutation step introduces small faults into the specification. All mutated implementation are first-order mutants of the specification. The type of errors is governed by a configurable set of mutation operators [57], which perform the actual mutation. Some typical mutation operators, supported by the Mothra system [61], are listed by Jia and Harman [57]. The operators available for mutating action systems are given in Table 5.1. Most of them correspond to a restricted portion of the operators offered by Mothra [61]. The mutation component is designed to be extensible through the implementation of further mutation operators. However, mutation is restricted to the *init*-block, to the guard of an action and to the state update of an action. This restriction is placed on mutation mainly for two reasons:

- Focus on first order mutants: Other types of mutations may require further mutations.
- Applicability of the conformance check: For the conformance check to be applicable, both specification and implementation must define the same set of actions using equivalent parameter lists. Hence, mutations must neither affect parameter lists, nor add or delete actions.

Type of Tests. Model-based mutation testing is a fault-based approach to test case generation. Usually tests are generated from counterexamples for conformance between the specification model and some mutant model. As a result, a test generated for some faulty mutant model M would detect a fault in an implementation of M. Hence, the tests generated by this approach test for the absence of faults. Since only specific faults specified through mutation operators are checked, the famous statement of Edsger Dijkstra ”Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” [41] remains true in general.

Nevertheless, it is an encouraging observation that testing can guarantee the absence of faults. But there are limitations that apply. If it is not possible to model a fault then its absence cannot be guaranteed either. A fault may be impossible to model if there exists no corresponding mutation operator, or if the fault cannot be expressed at the level of abstraction of the specification model. Furthermore, the absence of faults can actually only be guaranteed for deterministic systems. In order to guarantee the absence of faults in non-deterministic systems it is necessary to assume *fairness* over the non-deterministic choice of actions. It needs to be assumed that all non-deterministic branches are eventually executed, if the generated tests are performed sufficiently often. However, this is actually a common assumption in black-box testing.

Concluding the statements above, model-based mutation testing aims to show absence of faults, thus it is rather a falsification approach than a verification approach. The type of errors, which are checked by the test cases shall be revisited in the following. Essentially they are given through the mutation operators that are applied.

However, the coupling effect, which is generally assumed to hold, states that simple errors are coupled to complex errors such that test data that finds simple errors also finds complex errors [57, 71]. Hence, generated test cases may also detect errors which have not been modelled explicitly. The assumption that tests generated for fine-grained faults will also cover coarse-grained faults has also been made for one case study performed by Aichernig et al. [4] in the context of model-based mutation testing. Its validity has been supported by experimental results.

Operator Name	Description
replace Boolean by $True$	replaces a Boolean expression by the Boolean constant $True$
replace Boolean by $False$	replaces a Boolean expression by the Boolean constant $False$
replace relational operator	replaces a relational operator defined for numerical values by $<$, $\leq$, $>$ or $\geq$
invert addition	replaces an addition expression for a subtraction and vice versa
increment	adds one to a constant or a variable
decrement	subtracts one from a constant or a variable
change enumeration constant	replaces an enumeration constant for another enumeration constant of the same type
negate Boolean	negates a Boolean expression
remove function/operator application	replaces the application of a function/-operator with one of its parameters (currently implemented for a restricted set of functions/operators)

Table 5.1: The action system mutation operators

A specific type of fault, which is not explicitly modelled but checked by tests generated through **sioco** conformance checking is *ignoring of inputs*. It has been stated in Section 4.4 that angelic completion can be seen as a mutation. Hence, through the application of angelic completion the effect of ignoring certain inputs is checked. It should be noted that the set of ignored inputs is affected by mutation. A mutation may for instance strengthen a guard such that a lower number of parameters is accepted. More inputs would be ignored through angelic completion as a consequence of such a mutation,.

5.1.2 Test Execution Phase

The testing strategy applied in this thesis is model-based, which means that the model can be used to generate tests and also to infer test verdicts [80]. However, symbolic test cases, as produced by the described process, do not state how to choose verdicts. Such verdicts are rather assigned through an online-testing phase driven by both test case and model. During this phase the test case defines stimuli that shall be sent to the SUT, while the model serves as the test oracle. This testing process shall be discussed in more detail below.

Overview

In the test execution phase, abstract symbolic test cases consisting of a trace and a formula are interpreted by a test driver and executed on an SUT. The traces specify which actions need to be executed and the formula specifies the conditions, which need to be satisfied by action parameters. After the execution of the last action of a trace, one last observation of the SUT's operation is evaluated.

Since this way of executing tests does not make any assumptions about the kind of formula used in a test case, it is not only applicable for model-based mutation testing, it may be used for other types of tests as well. In order to execute boundary value tests for instance, the formula could specify that boundary values should be chosen for parameters of input actions.

Concerning model-based mutation testing, the formula in addition to the trace directs the test execution to states in which non-conformance would be observed, if a mutated specification was implemented by the SUT. Hence, the goal of the test case execution is to reach such a state and to check if the SUT conforms to the model in this state. However, such a state may not be reached, if the SUT performs an action not expected by the test case. This may happen if the SUT does not conform to the specification, which leads to a *fail*-verdict. But it may also happen if the SUT conforms to the specification, because action systems allow for underspecification, thus a system may non-deterministically produce any output of a set of enabled outputs. Such a case would lead to an *inconclusive*-verdict. An *inconclusive*-verdict is used to signal that although non-conformance was not detected, the goal of the test was not fulfilled. The unsafe state was not reached during test case execution. As noted above, the verdicts are inferred via an interpretation of the model. The steps and artefacts involved in the testing process shall be shortly described:

Parsing, Type-Checking and Translation of Specification. The specification must be executed symbolically because it is needed for assigning verdicts. As a result, it must be parsed, type-checked and translated. These steps are performed identically in the test case generation phase.

Parsing of Test Cases. In this step, the trace of actions and the condition associated with a test case are parsed.

Symbolic Execution of Traces. This precomputation step is performed before each test case. It determines all compound symbolic states reachable by executing the trace of actions without considering the condition associated with the test case.

Test Bridge. This component implements a communication channel used to exchange messages between test driver and SUT.

Test Case Execution. During test case execution, concrete parameter values are chosen for input actions, which are executed on the SUT. Additionally observations of the SUT are evaluated and thereby checked for conformance.

Verdict. One of three different verdicts may be assigned:

Fail. A *fail*-verdict is assigned, if the test case execution detected non-conformance of the SUT.

Inconclusive. An *inconclusive*-verdict is assigned, if the test case execution did not detect non-conformance, but did not reach a concrete unsafe state after the execution of the trace either.

Pass. A *pass*-verdict is assigned, if the test case execution did not detect non-conformance and reached a concrete unsafe state after the execution of the trace.

A concrete unsafe state is a product state interpretation, which satisfies the non-conformance condition formed with respect to the specification and a non-conforming mutant. Hence, a concrete unsafe state may be reached after simultaneously executing a sequence of actions with concrete parameters on specification and mutant. Consequently, an SUT implementing the mutant may show non-conforming behaviour if such a state is reached.

$$
\begin{aligned}
&\textbf{structure}\ TestBridge\{\\
&\ sendInput(event : \Lambda_I \times \mathcal{P}(\mathcal{I} \times \mathfrak{U})) : Unit\\
&\ receiveOutput() : (\Lambda_U \cup \{\delta\}) \times \mathcal{P}(\mathcal{I} \times \mathfrak{U})\\
&\}
\end{aligned}
$$

Figure 5.2: Specification of the test bridge signature

Test Driver Structure and Execution Algorithm

Since the test cases are symbolic, the test driver, which executes those test cases, needs to deal with symbolic computations as well. In the following, a high-level view of a test driver based on symbolic execution will be presented. This test driver has prototypically been implemented in the course of this thesis, but test case execution will not be evaluated in depth. The effectiveness of test cases generated through model-based mutation testing with an underlying **ioco** checker has for instance been examined by Aichernig et al. [4].

Roughly speaking, it is necessary to deal with two situations during test case execution:

- an input needs to be chosen and sent to the SUT,
- or an observation of the SUT is made and needs to be evaluated. An observation may either be an output action or quiescence, the absence of outputs.

This simplified view ignores a subtle problem related to so-called mixed states, which needs to be addressed for testing performed in a production setting. Mixed states are states in which both input and output actions are enabled [4]. Nevertheless, based on this view, it is possible to reason about the structure of the test driver.

It is obviously necessary to somehow communicate with the SUT. In order for the test driver to be generally applicable, this communication is kept abstract and should be implemented with respect to the actual SUT. Hence, a test bridge responsible for communication is introduced and its signature is defined as in the pseudo-code given in Figure 5.2. It is a structure offering two methods, one for receiving outputs and one for sending inputs. Since the test bridge handles concrete events, an event is a pair consisting of an action label and a set of key-value pairs, where the value defines the concrete value of the parameter given by the key. The type *Unit* corresponds to a `void`-return type in C-like languages.

For the remaining discussion, a symbolic test case $tc = (\sigma, \xi)$ is assumed to be given, where $\sigma = \sigma_1 \cdot \sigma_2 \cdots \sigma_n \in (\Lambda \cup \{\delta\})^*$ is a trace of actions and quiescence observations and $\xi \in \mathfrak{F}(\widehat{\mathcal{I}}_n \cup \mathcal{I})$ is an associated condition. More concretely, ξ must satisfy $\mathsf{free}(\xi) \subseteq \bigcup_{i \leq n} r_i(\mathsf{para}(\sigma_i)) \cup \mathcal{I}$. This means that the constraints, which are represented by ξ, mention only indexed parameters, which occur on the trace σ and non-indexed parameters. In the case of test cases derived via the **sioco** conformance checker, the non-indexed parameters are the parameters of the observation leading to non-conformance.

Now it is necessary to discuss when and how inputs need to be chosen and sent. Analogously, it shall be discussed when it is necessary to wait for outputs and how to evaluate them. The question as to when an input should be sent and when an observation should be evaluated is actually straight-forward to answer. During test case execution, each step σ_i shall be executed one after another. Hence, if σ_i is an input action, an input needs to be sent, otherwise an output is expected to be received via the test bridge. Note that outputs implicitly include quiescence as well, although strictly speaking, it is the absence of outputs.

Recall the goal of test case execution for the choice of inputs. The goal is to reach a concrete unsafe state and to observe conforming behaviour along the way and in this state. Consequently, σ_i should be sent to the SUT if it is an input and the parameter values should be chosen such that they satisfy ξ,

as ξ encodes a condition satisfied by a (symbolic) unsafe state. If the current step σ_i in the trace is an observation, the test driver should use the test bridge to receive an output, which may also be quiescence. The evaluation of the received output, may lead to three different outcomes:

1. If the received output label is equal to σ_i and if the parameter values satisfy condition ξ, the test execution continues with σ_{i+1}.
2. If the first condition does not hold, but the received output including the parameters is allowed by the specification, the test execution stops with an *inconclusive*-verdict. This means that the test goal of covering the injected fault has not been achieved.
3. If the received output including the parameters is not allowed by the specification, then the test execution stops and a *fail*-verdict is issued.

After all steps from σ_1 to σ_n have successfully been performed, the test driver should wait for another output and check if it is allowed by the specification. If it is allowed, a *pass*-verdict is issued, otherwise a *fail*-verdict is issued. This last step is performed, as an SUT implementing the mutated specification would show non-conforming behaviour in the state reached after the n^{th} step.

Before these considerations are formalised through the definition of a test driver algorithm, it should be noted that there may be interdependencies between steps. This means that parameters of an action σ_i may place constraints on the parameter values of an action σ_j later in the trace, that is, $j > i$. As a result, the condition ξ should actually be updated after each step σ_i. The parameters $\mathsf{para}(\sigma_i)$ should be set to be equal to the actual concrete parameter values. An implementation of a symbolic test driver is given in Algorithm 7. It combines the precomputation step *Symbolic Execution of Traces* and the actual test execution in one function to ease comprehension.

The algorithm takes a trace σ and an associated condition ξ as well as a specification action system and a test bridge as input. An initialisation is performed in Lines 5 to 7. This includes the symbolic execution of a τ-closure setting the initial state of the specification. Afterwards, a loop over all action labels in σ is performed (Lines 8 to 32).

Inputs are executed in Lines 9 to 16, whereby concrete parameters for the currently processed input σ_i are chosen according to condition ξ and sent to the SUT together with σ_i (Lines 11 to 13). Additionally, σ_i is symbolically executed with the specification action system (Line 10). Furthermore, the concrete parameter values sent to the SUT are fixed in both the condition ξ and the symbolic state of the specification (Lines 14 to 16).

Outputs are observed in Lines 18 to 30. They are only expected when the currently processed action label σ_i is an output. In Line 19, the test driver waits for an output from the SUT with concrete parameter values. Similarly to inputs, outputs are also executed symbolically (Line 20) and the concrete parameters received from the SUT are set in both ξ and the symbolic state of the specification (Lines 20 to 22). If the received output is not expected by the trace σ and the condition ξ, it is checked whether the output is allowed by the specification (Lines 23 to 30). Disallowed outputs lead to *fail*-verdicts while unexpected, but allowed outputs lead to *inconclusive*-verdicts (Lines 25 to 29).

Eventually, when all actions in σ have been processed, an unsafe would be reached. In order to check whether the SUT implements the mutated specification, one last output from the SUT is received and analysed (Lines 33 to 42). Allowed outputs lead to *pass*-verdicts, whereas outputs disallowed by the specification lead to *fail*-verdicts. Since quiescence is considered to be an output, it is always possible to receive an output without providing additional stimuli.

Lines 25 and 38 contain checks of path conditions of specification states. These lines highlight why it is necessary to symbolically execute the specification in parallel to the test execution: symbolic test cases neither state how to interpret the last output (Lines 33 to 42) nor how to interpret unexpected outputs (Lines 23 to 30).

Algorithm 7 The symbolic test execution algorithm.

function ADDCONSTRAINT($state$,$constraint$)
  return $\{(\varphi \wedge constraint, \rho) | (\varphi, \rho) \in state\}$
end function
function TESTEXEC($specification$,σ,ξ, $testBridge$)
  $\langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, trans\_rel \rangle \leftarrow specification$
  $trans\_rel \leftarrow trans\_rel \cup \{(\delta, \Delta, \mathsf{id})\}$
  $state \leftarrow tau_{cl}(\{(\top, \iota)\})$
  for $i = 1$ **to** $length(\sigma)$ **do**
    if $\sigma_i \in \Lambda_I$ **then**
      $state \leftarrow tau_{cl}(exec(state, guard(\sigma_i, trans\_rel), update(\sigma_i, trans\_rel)))$
      $model \leftarrow getModel(\xi)$
      $concPara \leftarrow \{(p, getValue(model, p)) | p \in \mathsf{para}(\sigma_i)\}$
      $testBridge.sendInput((\sigma_i, concPara))$
      $equalConstraint \leftarrow \bigwedge_{(p,v) \in concPara} r_i(p) = v$
      $state \leftarrow addConstraint(state, equalConstraint)$
      $\xi \leftarrow \xi \wedge equalConstraint$
    else
      $(\sigma_{rec}, parameter) = testBridge.receiveOutput()$
      $state \leftarrow tau_{cl}(exec(state, guard(\sigma_{rec}, trans\_rel), update(\sigma_{rec}, trans\_rel)))$
      $equalConstraint \leftarrow \bigwedge_{(p,v) \in parameter} r_i(p) = v$
      $\xi \leftarrow \xi \wedge equalConstraint$
      $state \leftarrow addConstraint(state, equalConstraint)$
      if $\sigma_{rec} \neq \sigma_i \vee \xi$ is not satisfiable **then**
        $pc \leftarrow \bigvee_{(\varphi,\rho) \in state} \varphi$
        if pc is not satisfiable **then**
          return FAIL
        else
          return INCONC
        end if
      end if
    end if
  end for
  $(\sigma_{last}, parameter) = testBridge.receiveOutput()$
  $state \leftarrow tau_{cl}(exec(state, guard(\sigma_{last}, trans\_rel), update(\sigma_{last}, trans\_rel)))$
  $equalConstraint \leftarrow \bigwedge_{(p,v) \in parameter} r_{n+1}(p) = v$
  $state \leftarrow addConstraint(state, equalConstraint)$
  $pc \leftarrow \bigvee_{(\varphi,\rho) \in state} \varphi$
  if pc is not satisfiable **then**
    return FAIL
  else
    return PASS
  end if
end function

As noted before, mixed states might pose a problem in practice. In our current applications, which are discussed in Chapter 7, we can safely assume synchronous communication, thus the test driver can block outputs from the SUT, if inputs should be sent [12]. Hence, the problem of mixed states is solved and the implementation of the test bridge is actually simple. However, this assumption may be too strong in some cases [35]. In such cases, communication may be performed asynchronously and outputs may arrive at any point in time. As a result, questions regarding the evaluation of outputs arise and the test bridge implementation should for instance use message queues. However, a thorough discussion of these issues is beyond the scope of this thesis.

This section discussed the main application area of the conformance check, which is model-based mutation testing. In the first part, the test case generation was discussed while the second half focused on test case execution. Although there is an explicit test-case generation phase, the testing approach cannot be classified as offline-testing. It is rather a hybrid approach which combines offline- and online-testing. In an offline-phase, symbolic test cases are generated, which are run subsequently. Since those test cases do not contain conditions for assigning verdicts, a symbolic execution of the action system specification must be performed in parallel. Hence, the actual testing process can be seen as online-testing.

In the following, the second application area of the conformance check will be discussed.

5.2 Model-Checking

Some of the steps performed during test case generation are specific to model-based mutation testing based on first-order mutants of a specification, but some are also more generally applicable. The mutation step, the test case generation step and the syntactic mutation analysis are specific to the model-based testing approach and for this reason are shown with dashed borders in Figure 5.1a. The other steps need to be performed for **sioco**$_d$ conformance checks between action systems in general. Since the **sioco**-conformance check performed during test case generation can be configured such that it does not make assumptions about the structure of implementations, it may be used for arbitrary action systems representing implementations.

Hence, the conformance check is more generally applicable and is not limited to test case generation. It may for instance be used to perform bounded model checking for the property of **sioco**-conformance [20]. If the conformance check is performed for a sufficiently large exploration depth of the product graph it is actually possible to perform a complete unbounded verification of **sioco** conformance and thus also **ioco** conformance. Sufficiency of exploration depth with respect to a maximum exploration depth d can be characterised as follows:

Proposition 5.1 (Sufficiency of Product Graph Exploration Depth).
Let $\mathcal{AS}_S$ be an action system representing a specification, let $\mathcal{AS}_P$ be an action system representing an implementation, let $SP = \mathcal{AS}_P \times_{sioco_{det}} \mathcal{AS}_S(d+1)$ be a deterministic symbolic synchronous product graph, and let q_{init} be its initial state. If SP does not contain fail-states and if for all sequences of actions and quiescence observations $\sigma \in \Lambda_\delta^*$, such that $q_{init} \stackrel{\sigma}{\Rightarrow} q$, it holds

- *that $length(\sigma) \leq d$*
- *or that there exists a q_{eq} and a $\sigma_{eq} \in \Lambda_\delta^*$ such that $q_{init} \stackrel{\sigma_{eq}}{\Longrightarrow} q_{eq}$, $q \equiv_{prod} q_{eq}$ and $length(\sigma_{eq}) < length(\sigma)$,*

then d is sufficiently large. It follows that $\mathcal{AS}_P$ $\mathbf{sioco}_d$ $\mathcal{AS}_S$ for all d.

The two conditions specify situations, in which it is not necessary to explore traces any further, because it would not uncover non-conformance. Consequently, if either of the conditions are met by all traces in a product graph, the exploration depth does not need to be increased.

Another condition for sufficiency can be given by considering the optimised **sioco** checking algorithm. If the search performed during **sioco** checking stops before reaching the maximum search depth d then $d - 1$ is sufficiently large. The depth-first search backtracks before reaching depth d if a state is reached,

- in which Product Graph Pruning as described in Section 4.2 would take effect, which corresponds to the second condition,
- or if a state is reached of which all post-states are unsatisfiable. Such states satisfy the first condition.

One intended application area of model-checking for **sioco** conformance has been discussed in the introduction in Section 1.5. The **sioco** conformance check could be applied during the stepwise development of test models. It would thereby be possible to ensure **ioco** conformance between each pair of consecutive models. As a result, refinements would be guaranteed to not introduce unwanted output behaviour.

6 Implementation

The implementation of the **sioco** conformance checker and the corresponding environment will be discussed in this chapter. A more in-depth description of selected components involved in the model-based testing process and in the model-checking process will be given for this purpose.

While the description of the **sioco** checking algorithm given in Section 3.2 aimed to be generally applicable, the descriptions given in this chapter mainly focus on technology-specific issues and details relevant to the implementation. As a result, some parts of the following discussion will only be relevant for implementations using SMT-solvers for checking satisfiability of first-order formulas and more specifically for implementations using the SMT-solver Z3 implemented and maintained by Microsoft [37]. This restriction is actually not severe, as Z3 is a popular SMT-solver, often used for testing and verification [22]. Furthermore, it would be possible to extend the implementation in order to support several solvers, as there exists an input language called SMT-LIB v2 [15] which is supported by many solvers.

Other technology-specific decisions are the choice to use the Scala language for the implementation and the decision to use the Java-API to communicate with Z3. These decisions, however, are not a major limiting factor for a reimplementation, as the only requirement for the implementation language is that it must be possible to interact with Z3. Since Microsoft offers a C-API, this requirement is fulfilled by a wide variety of commonly used programming languages.

While focusing on the most important technology-specific issues, this chapter will ignore details like the class structure of the Scala-project, which implements the mutation-based test case generator.

6.1 Type System

The syntax definition given in Section 2.3.1 allows the utilisation of three different kinds of data types, namely integer range types, enumeration types and the Boolean type. In order to enable more convenient modelling of complex systems, an extension to this restricted type system was developed in joint work with Benedikt Maderbacher. Additionally to the already mentioned kinds, the extended type system allows for the definition of complex data types parameterised by other types. The data types available through the type system extension comprise:

Set Data Types: sets of values of the same type

Map Data Types: sets of key-value pairs of the same type

Record Data Types: sets of key-value pairs of fixed size and possibly non-uniform value type

All types can be defined using the adapted nonterminal symbol $TYPE$ given in Figure 6.1 and are equipped with several standard functions like for instance a set membership check for set data types. In addition to these complex data types, two additional predefined types have been introduced as well: Int and $Real$ modelling unbounded integers and unbounded real numbers respectively.

$$TYPE ::= \texttt{[}integer\texttt{..}integer\texttt{]} \mid \texttt{[}\overline{capid\texttt{|}capid}\texttt{]} \mid$$
$$\texttt{Set[}ty\texttt{]} \mid \texttt{Map[}ty\texttt{,}ty\texttt{]} \mid \{\overline{id : ty}\}$$

Figure 6.1: The extended $TYPE$ nonterminal symbol, which was first defined in Figure 2.3. The last option defines a record data type.

A type checker has been implemented which supports this extended type system. It places a restriction on the types of action parameters, which forbids the combination of integer range data types with complex data types. The reason for the introduction of this restriction is that the combination would require complex guards. For instance, a guard of an action taking sets of integers in some range R would need to state that all set elements must be in R. Hence, (nested) universal quantification would be needed in translated guards, which would likely degrade performance significantly.

Both sets and maps have been implemented via the *array*-sort provided by Z3 [36]. Arrays as provided by Z3 essentially allow to store key-value mappings of arbitrary type. Furthermore, records and maps also make use of algebraic data types.

A set in this context, is an array mapping from the type of elements to the Boolean type. In order to denote the presence of an element e, a mapping from e to $true$ is inserted. The element can be removed by inserting a mapping from e to $false$, thus overwriting the mapping to $true$. Through setting the default array value to $false$, it is possible to state that a set is initially empty.

Maps with key-value types (k, v) are implemented as arrays mapping from k to an algebraic data type w. The data type w has two constructors, one constructor $some$ which essentially wraps an element of type v and one constructor $none$ without parameters (this is inspired by the `Option`-data type provided by Scala). The default value of a map is set to $none$, thus a mapping from a key ke to $none$ denotes the absence of elements with key ke. Conversely the presence of a key-value pair (ke, ve) is denoted through the insertion of a mapping from ke to $some(ve)$ into the array.

Records are implemented as special algebraic data types, which declare one constructor with one parameter per record element. Furthermore, one accessor is declared for each record element.

Experiments performed during implementation revealed a defect in Z3's handling of arrays[1]. The bug has been reported and subsequently fixed.

6.2 Parsing and Mutation

The implementation of the parsing component has actually merely been adapted in the course of this thesis. It has originally been developed by Benedikt Maderbacher during his Bachelor's thesis under the supervision of Bernhard Aichernig and thus will not be discussed in detail. The implementation uses the parser combinator library provided by Scala [70] and creates AST-representations of action systems.

Based on these ASTs, mutations are performed through the application of the mutation operators listed in Table 5.1. As can be seen from the table, the focus lies on models using simple data types, as only the last mutation operator directly affects values of complex data types. Nevertheless, the other mutation operators also affect models using complex data types.

Since the implementation of the mutation component did not require a major technology-specific decision, it will not be discussed in great detail either. However, it should be noted that representing the AST through Scala's *case classes* allowed a concise recursive implementation.

6.3 Translation

The translation component maps ASTs of action systems to their SMT-representations, that is, it maps to Java-objects which in turn reference objects provided by Z3. Since Z3 offers a rich functionality, there exist direct translations for all expressions in the action system language. Nevertheless, two aspects of the translation shall be discussed.

[1] Issue #173: `https://z3.codeplex.com/workitem/173` (last visit: 23.11.2015)

6.3.1 Implicit Extension of Guards

In order to ensure that only valid values are assigned to variables of integer range types, action guards are implicitly extended. These extensions constrain the possible values for both state and parameter variables and are added to the original guards via conjunction. More precisely, for a variable v of type $[a..b]$ the guard of an action act is extended by

- $v \geq a \wedge v \leq b$ if v is a parameter variable
- and by $e \geq a \wedge e \leq b$ if v is a state variable and there exists an assignments $v := e$ in act.

Similar constraints are created for values stored in complex data types, such as sets containing integers in a given range. As a result, mutations of assignments may affect guards and thereby may disable the execution of actions. The syntactic mutation analysis presented in Section 4.3 takes this into account by comparing the syntactic structure of translated and extended guards.

6.3.2 Choice of API

Several factors were considered for the choice of the way of interacting with Z3. Actually three different approaches have been used throughout the development until the Java-API was finally chosen. In the first stage of development, expressions were translated to SMT-LIB strings [15] and parsed using Scala^{Z3} [62]. Since SMT-LIB is a standard format, it allows the use of different SMT-solvers. This approach was inspired by the Scala implementation discussed by Aichernig et al. [7].

However, as all other formulas such as path conditions were stored in the SMT-LIB format as well, it turned out that a significant amount of computation time was used for parsing and string manipulation. As efficiency was considered to be more important than exchangeability of SMT-solvers, the Scala^{Z3}-API was used more extensively in the second stage of development. In this stage, objects representing SMT-terms and formulas were used rather than SMT-LIB strings.

This library, however, suffers from a serious drawback. The memory management is not suitable for symbolic execution. For symbolic execution it is necessary to create a large number of objects representing AST-nodes of SMT-formulas, which is done using objects of a class called `Z3Context`. In the current implementation, a `Z3Context`-object o holds references to all created objects until the deletion of o is performed. Stated differently, the library prevents the garbage collector from collecting unused objects. As a result, the symbolic execution cannot be performed because an excessive amount of RAM would be needed. Hence, the library is only applicable if the problem to be solved can be partitioned into several smaller subproblems, which all use different `Z3Context`-objects.

For this reason, the final implementation uses the Java-API provided through the git-repository[2] which also hosts the source code of the Z3 SMT-solver. This API implementation makes use of the garbage collector and also releases the memory allocated by native code if unused objects are garbage-collected. Beside this difference both API implementations are similar in terms of functionality.

6.4 Conformance Check

The conformance checking algorithm has been implemented as discussed in Section 3.2 and Chapter 4, so only implementation relevant issues shall be discussed. Most of these issues arise from the fact that checking equivalence of symbolic states is in general a difficult task, which may prevent the application of some optimisations. As a result, configuration parameters have been introduced to be able to selectively turn off optimisations and to select strategies for checking equivalence. Another reason to disable some of the optimisations, is to be able to perform conformance checks between arbitrary action systems.

[2] `https://github.com/Z3Prover/z3` (last visit: 12.11.2015)

6.4.1 Equivalence Checks

As noted above, equivalence checks are in general difficult to perform. This is caused by the fact that checking equivalence involves deciding validity of quantified first-order formulas. While Z3 is able to check satisfiability of many formulas containing quantifiers [36] (Ge and de Moura for instance list decidable fragments of first-order logic modulo theories [48]) and efficient quantifier elimination procedures are being developed [21], performing equivalence checks may still be impractical. This has been concluded from experiments, which have shown that inefficient equivalence checks may result in a performance loss, which can not be mitigated by pruning made possible through those checks. Equivalence checking may also be inefficient in the sense that it is infeasible for a large number of symbolic states of a model. In these cases, infeasible checks introduce a significant computational overhead and do not allow to prune traces. Hence, disabling optimisations which build upon symbolic state equivalence may make sense.

Equivalence Checks using SMT-solvers. The basic implementation of equivalence checks by means of SMT-solving shall now be described. Let $\eta = (\varphi, \rho)_i$ and $\eta' = (\varphi', \rho')_j$ be two symbolic indexed states. Recall the first condition of symbolic state equivalence given in Definition 2.19, which corresponds to the state inclusion condition for STSs [47]:

- $[\![\eta]\!] \subseteq [\![\eta']\!]$ if for all $\zeta \in \mathfrak{U}^{\mathcal{V}}$ and a $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}_i}$ such that $\zeta \cup \upsilon \models (\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi)$ there exists a $\upsilon' \in \mathfrak{U}^{\widehat{\mathcal{I}}_j}$ such that $\zeta \cup \upsilon' \models (\bigwedge_{x \in \mathcal{V}} x = \rho'(x) \wedge \varphi')$

This condition needs to be encoded as first-order formula, which shall be based on the following four observations:

- The existential quantification of valuations of parameter variables corresponds to the existential quantification of all parameter variables.
- A conditional sentence can be modelled via implication.
- The condition must hold for all $\zeta \in \mathfrak{U}^{\mathcal{V}}$, which may either be modelled using universal quantification, or by checking if the unquantified formula is a tautology. The latter strategy shall be used in order to avoid using additional quantifiers.
- It is only possible to check satisfiability of a formula $\gamma \rightarrow \psi$ with SMT-solvers, but not validity. Stated differently, it is not possible to check if some formula is a tautology. Nevertheless, validity of $\gamma \rightarrow \psi$ may be inferred if $\neg(\gamma \rightarrow \psi) = \gamma \wedge \neg\psi$ unsatisfiable, which can be checked with SMT-solvers.

Hence, in order to show that $[\![\eta]\!] \subseteq [\![\eta']\!]$, it shall be checked if

$$\left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \left(\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi \right)\right) \wedge \neg \left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \left(\bigwedge_{x \in \mathcal{V}} x = \rho'(x) \wedge \varphi' \right)\right)$$

is unsatisfiable. Since for equivalence, it must also hold that $[\![\eta']\!] \subseteq [\![\eta]\!]$, the formula

$$\left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \left(\bigwedge_{x \in \mathcal{V}} x = \rho'(x) \wedge \varphi' \right)\right) \wedge \neg \left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \left(\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi \right)\right)$$

needs to be shown to be unsatisfiable as well. Alternatively, equivalence may be checked directly, thus it may be shown that

$$\neg \left(\left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \left(\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi \right)\right) \Leftrightarrow \left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \left(\bigwedge_{x \in \mathcal{V}} x = \rho'(x) \wedge \varphi' \right)\right) \right)$$

is unsatisfiable. However, the implementation uses two separate checks, because they have shown better performance in experiments.

An equivalence condition for compound symbolic states can analogously be derived as follows. Let κ_i and κ_j be two indexed compound symbolic states and let ξ_i and ξ_j be defined by:

$$\xi_i = \bigvee_{(\varphi,\rho)_i \in \kappa_i} \left(\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi \right)$$

$$\text{and } \xi_j = \bigvee_{(\varphi,\rho)_j \in \kappa_j} \left(\bigwedge_{x \in \mathcal{V}} x = \rho(x) \wedge \varphi \right)$$

$\kappa_i \equiv_{com} \kappa_j$ if

$\left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \xi_i\right) \wedge \neg \left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \xi_j\right)$ and $\left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \xi_j\right) \wedge \neg \left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \xi_i\right)$ are unsatisfiable

Finally, a similar equivalence condition for product states shall be derived. Let (κ_i, μ_i) and (κ_j, μ_j) be two indexed product states, where κ_i and κ_j are compound symbolic states of an implementation $\mathcal{AS}_P$ with state variables $\mathcal{V}_P$ and μ_i and μ_j are compound symbolic states of a specification $\mathcal{AS}_S$ with state variables $\mathcal{V}_S$. Let ξ_i and ξ_j be defined by:

$$\xi_i = \left(\bigvee_{(\varphi,\rho)_i \in \kappa_i} \left(\bigwedge_{x \in \mathcal{V}_P} x = \rho(x) \wedge \varphi \right) \right) \wedge \left(\bigvee_{(\varphi,\rho)_i \in \mu_i} \left(\bigwedge_{x \in \mathcal{V}_S} x = \rho(x) \wedge \varphi \right) \right)$$

$$\text{and } \xi_j = \left(\bigvee_{(\varphi,\rho)_j \in \kappa_j} \left(\bigwedge_{x \in \mathcal{V}_P} x = \rho(x) \wedge \varphi \right) \right) \wedge \left(\bigvee_{(\varphi,\rho)_j \in \mu_j} \left(\bigwedge_{x \in \mathcal{V}_S} x = \rho(x) \wedge \varphi \right) \right)$$

$(\kappa_i, \mu_i) \equiv_{prod} (\kappa_j, \mu_j)$ if

$\left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \xi_i\right) \wedge \neg \left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \xi_j\right)$ and $\left(\overline{\exists}_{\widehat{\mathcal{I}}_j} \xi_j\right) \wedge \neg \left(\overline{\exists}_{\widehat{\mathcal{I}}_i} \xi_i\right)$ are unsatisfiable

As discussed above, executing those checks may decrease performance in some cases, so a configuration parameter `disableEquiv` has been introduced to disable equivalence checks. More concretely, if `disableEquiv` is set to true then $q \equiv q'$ will always yield $\bot$ without performing any checks. Consequently all optimisations which utilise equivalence checks do not work if `disableEquiv` is set. Another configuration parameter `tryQuantifierElimation` is available, to enable quantifier elimination. If this parameter is set to true and an equivalence checks fails, quantifier elimination is applied on the equivalence conditions and the equivalence check is performed again. The reason for performing quantifier elimination only after explicit configuration is that quantifier elimination causes a performance penalty if not needed.

Strategies. Z3 offers the possibility to define customised strategies for SMT-solving. These strategies allow the combination of reasoning engines based on application-specific needs [38]. Since equivalence checks constitute a different type of problem than other satisfiability checks, such as checks of path conditions for instance, experiments with strategies have been carried out. These experiments led to another further configuration parameter for choosing an equivalence check strategy. One of the following strategies can be chosen:

Default. The default SMT-solver.

Default with Timeout. The default SMT-solver with a configurable soft timeout. This is actually not a strategy, but could be implemented via a strategy.

Quantifier Elimination after Fail. This strategy applies the default SMT-solver and returns the result if it is successful. If the default solver is not successful, quantifier elimination followed by the default solver strategy is applied. The strategy can be expressed in SMT-LIB via: `(or-else smt (then qe smt))`.

Quantifier Elimination after Timeout. This strategy applies the default SMT-solver for a configurable amount of time and returns the result if it is successful. If it is not successful, quantifier elimination followed by the default solver strategy is applied. The strategy can be expressed in SMT-LIB via: `(or-else (try-for smt x) (then qe smt))` where `x` defines the timeout.

As the last two strategies incorporate quantifier elimination, the `tryQuantifierElimation`-parameter is only effective while using the first two strategies. It can be seen that there are redundancies considering the combination of strategies and the `tryQuantifierElimation`-parameter. They exist for evaluation purposes. Experiments involving these strategies showed that applying quantifier elimination after a timeout can help to significantly reduce computation time. This is caused by the fact that it may take very long for the default solver to fail. Motivated by the results, a similar configuration parameter has been introduced for the conformance check, as checking for quiescence also involves quantification.

During experiments involving the mentioned tactics, defects of Z3 have been detected and reported to the development team[3]. The defects affected the strategy *Quantifier Elimination after Timeout* and the `qe-sat`-tactic, which is not used anymore in the implementation. They caused wrong satisfiability check results as well as segmentation faults. Since then, the issue has been resolved and newer versions of Z3 do not show the detected erroneous behaviour.

6.4.2 τ- Divergence

In Section 3.1.1, it was mentioned that the symbolic execution tree of divergent action systems may be of infinite size. In general, divergence may either result from loops of internal actions which reach some set of equivalent symbolic states repeatedly or from executable sequences of internal actions, which reach unbounded sets of concrete states when executed. The first condition is dealt with in Algorithm 3 by pruning, while the latter can only be fulfilled if unbounded types are used for state variables, but cannot be handled effectively by the algorithm. As a result, Algorithm 3 may not terminate, if equivalence checks need to be disabled or if the second condition for divergence is fulfilled. To counter this problem, a restriction on the applicability of the conformance shall be placed:

Remark (Restriction on Convergence of Specifications). The non-conformance check may only be performed for

- convergent specifications
- or for divergent specifications for which
  - there exists an upper bound on the number of concrete states reachable by executing internal action sequences
  - and for which it is possible to efficiently perform symbolic state equivalence checks.

Example 6.1 (Divergent Action System).
Let $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ be an action system using unbounded integers as data, where $\mathcal{V} = \{x\}$, $\mathcal{I} = \{\}$, $\Lambda_I = \{?input\}$, $\Lambda_U = \{!output\}$, $\iota = \{x \mapsto 0\}$ and $\rightarrow = \{(?input, x = 0, \{x \mapsto 1\}), (\tau, x \neq 0, \{x \mapsto x + 1\}), (!output, x = 3, \{x \mapsto 1\})\}$. The action system is divergent because there exists no upper bounded on the number of states reachable by executing internal actions. After executing the input action, the internal action may be executed twice and followed by an output. But it may be executed infinitely often as well and reach a new state after each execution by incrementing the state variable x. Hence, this action system should not serve as specification for test case generation.

[3] Issue #196: `https://github.com/Z3Prover/z3/issues/196` (last visit: 23.11.2015)

Divergence of Implementations. Although by the restriction given above, the τ-closure calculation will always terminate for specifications, it may not terminate for automatically generated mutants, especially if equivalence checks are disabled. Hence, it is necessary to place a limit on the number of recursive applications of τ_{cl}. An action system representing an implementation is considered to be divergent, if this limit is reached during the calculation of a τ-closure. In the implementation of the conformance check, this limit can be set using the configuration parameter `tauDivergenceLimit`. It needs to be set to a value high enough, such that the specification is not detected to be divergent. In most cases, mutants detected to be divergent will contain τ-loops and thus will actually be divergent, but a too low value for `tauDivergenceLimit` will lead to a wrong detection of divergence for action systems, which are actually convergent.

Furthermore, another configuration parameter `stopOnDivergence` can be set to $true$, to stop the conformance check if divergence is detected. In this case, the conformance check returns a trace to the current product state and the path condition of this state. Considering model-based mutation testing, this trace will detect an error in an SUT implementing the same error, thus also entering an infinite loop, assuming that divergence was detected because of a τ-loop.

However, there is also a more pragmatic reason for the introduction of the configuration parameter `stopOnDivergence`. Consider the detection of divergence during the calculation of a τ-closure because of a τ-loop. The τ-closure will contain at least n symbolic states, where n is given by the value of `tauDivergenceLimit`. In the next step, it is necessary to execute an action for all n states and possibly also calculate a τ-closure. If the τ-loop is the result of a weakened guard of an internal action then the number of states will most likely grow with increasing search depth, thus significantly slowing down the exploration. If on the contrary divergence is detected because of a too low value for `tauDivergenceLimit`, then the conformance check would be performed using a wrong implementation state, thus the check should be stopped as well.

6.4.3 Disabling Syntactic Mutation Analysis

By default, all optimisations of the conformance checking algorithm are enabled, but as some of them assume that the implementation is a first-order mutant, it is possible to disable all optimisations based on this assumption. This enables the utilisation of the conformance check for model-checking of the property of **sioco** conformance between arbitrary action systems.

7 Case Studies

This chapter introduces case studies carried out in the course of this thesis. The case studies mainly deal with models of three different systems and focus on mutation-based test case generation, with special regard to the efficiency of the symbolic approach to conformance checking. For each of the three systems, different models capturing the same requirements have been created. Consequently, action system modelling and the effects of modelling styles on test case generation will also be discussed in this chapter. However, a closer inspection of the effects of modelling styles in model-based mutation testing can be found in Tiran's Master's thesis [76].

The first two system models specify the behaviour of a simple supplier, which is also used as an example by Frantzen et al. [45]. Subsequently, the specification of a system measuring particle counts in exhaust gas will be discussed. In the next part of this chapter, different models of a car alarm system and corresponding experimental results will be presented.

Both the particle counter as well as the car alarm system served as a benchmark before. They have for instance been used to evaluate a mutation-based test case generator [58], which may be configured to use one of three conformance relations including **ioco**. As noted before, the concrete conformance checker for **ioco** has been implemented by the author of this thesis and will therefore be used for an evaluation of the efficiency of the symbolic approach. In order to evaluate the efficiency, a comparison between the measurement results of the concrete **ioco** checker experiments and the **sioco** checker experiments will be given. As mentioned in the introduction chapter, the symbolic **ioco** checker aims at improving upon the runtime of the concrete **ioco** checker. Consequently, the measurements focus on runtime.

This chapter will be concluded by a discussion of experiments performed with models using complex data types such as sets. Although the presented models are relatively simple, they are well-suited to demonstrate limitations of the current implementation and highlight issues, which need to be overcome.

Measurement Setup. All experiments were conducted on a computer equipped with 8 GB RAM and an Intel© Core™ i7 CPU running at 3.4 GHz. The runtime environment comprises the SMT-solver Z3 v4.4.1 and Scala v2.11 running in the Java Virtual Machine, with Java version 1.7.0_85, installed on Ubuntu Linux 14.04. The Java-API provided by Z3 was used for the communication with Z3. Furthermore, all available optimisations were applied unless otherwise noted.

Since the conformance check is the most time-consuming step in model-based mutation testing, the discussion will focus on this step. Corresponding experiments create a number of mutants and perform conformance checks between the specification and every mutant. In order to discuss runtimes, the mean, median, maximum and minimum duration of the conformance checks will be given. Additionally, the precomputation time, the number of created mutants and the total runtime of all conformance checks for one specification will be given as well.

The precomputation includes the creation of the symbolic execution graph and the calculation of reachable actions for the specification. Hence, it is performed only once during test case generation. Strictly speaking, precomputation includes parsing, type checking and translation, as well. Although these tasks are performed once per action system, the computation time spent for them is negligibly small and thus will be ignored.

7.1 Supplier

The first system is a simple supplier, which has been used as an example by Frantzen et al. to discuss STSs and IOSTSs respectively and their symbolic semantics [45]. Initially, the supplier was only used for testing, but it also demonstrates that there exists a simple translation from IOSTSs to action systems, as long as each observable action label is used at most once to label a transition. Furthermore, it can be

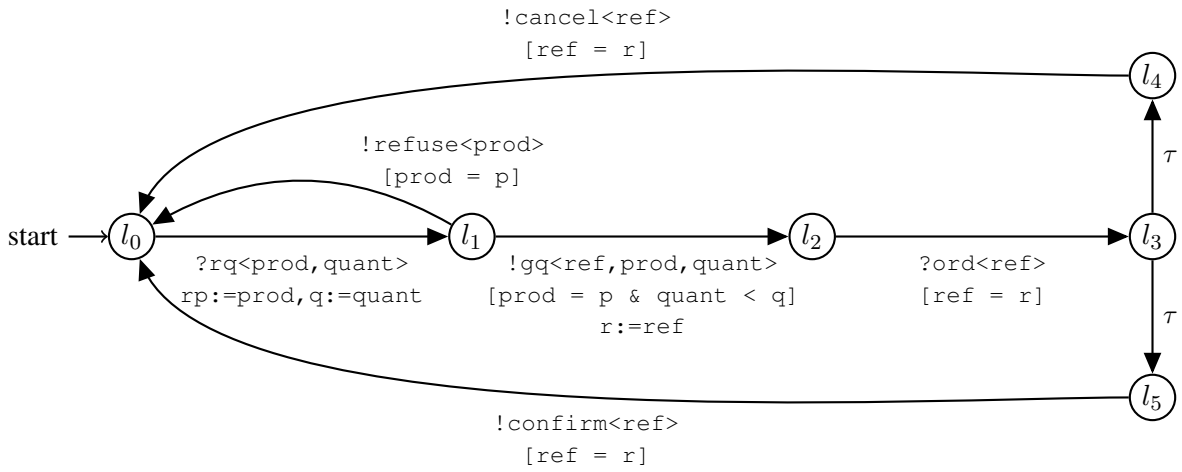


Figure 7.1: Adapted version of the supplier STS [45], containing an additional observable action `refuse`.

observed that the presence of internal actions has a negative impact on performance and that the symbolic conformance checker is well-suited for this kind of models.

7.1.1 Specification

An adaptation of the STS-model is given in Figure 7.1. Each transition is defined by at most three lines, where the first line defines the action label (gate) and its parameters (interaction variables), the second defines its guard enclosed in square brackets and the last line defines its state update. Empty state updates and guards equivalent to $\top$ are omitted. The model allows to request quotes for a given quantity of a product through the action `?rq`. Afterwards the quote may be granted through the action `!gq`, but only for a lower quantity, or it may be refused through the action `!refuse`. If granted, the quote can be ordered using `?ord`. For an order, the system chooses non-deterministically between a cancellation through `!cancel` and a confirmation of the order through `!confirm`.

Transformation into an Action System

The transformation of this model into an action system is straight-forward and involves the following steps:

- Creation of integer range data types for the different variables.

  This is actually optional, because the implementation offers a built-in unbounded integer type, which could be used. But it eases testing of the symbolic conformance check, as it is possible to also check **ioco** concretely if the state space is bounded. This allows to compare conformance check results of both approaches.
- Definition of one state variable per state variable of the STS[1].
- Creation of an enumeration data type `Location`, defining one constant per location.
- Definition of another state variable `currentLocation` of type `Location`, which is initialised to the initial location label.
- Definition of an appropriate initialisation of all state variables.
- Each transition t is transformed into an action, such that

[1] State variables of STSs are actually called *location variables*. However, the term *state variable* is rather used to avoid confusion with the variable `currentLocation`.

- the constraint `currentLocation == source location of` t is added to the action guard via conjunction,
- the assignment `currentLocation := sink location of` t is added to the state update
- and the transition gate and interaction variables are transformed into an action label and parameters respectively. The transition gate of observable transitions can be used directly, but for each unobservable transition, a unique action label must be defined. This is because all action system labels must be different on the level of concrete syntax in the version of action systems considered in this thesis.

The action system model of the supplier is given below:

```
def SimpleSupplier
{
  types {
    Quantity = [1..100000];
    ProductID = [1..10000];
    RefID = [1..20000];
    Location = [L0 | L1 | L2 | L3 | L4 | L5];
  }
  state {
    rp : ProductID;
    rq : Quantity;
    currentLocation : Location;
    refNr : RefID
  }
  init {
    rp := 1;
    rq := 1;
    currentLocation := L0;
    r := 1
  }
  actions {
    ?rq(prod:ProductID, quant : Quantity) if currentLocation == L0 then {
      rp := prod;
      rq := quant;
      currentLocation := L1
    };
    !gq(prod:ProductID, quant : Quantity, ref : RefID)
      if currentLocation == L1 && rp == prod && quant < rq then {
      rp := prod;
      rq := quant;
      currentLocation := L2;
      r := ref
    };
    !refuse(prod:ProductID) if currentLocation == L1 && rp == prod then {
      currentLocation := L0
    };
    ?ord(ref : RefID) if currentLocation == L2 && r == ref then {
      currentLocation := L3
    };
    chooseCancel() if currentLocation == L3 then {
      currentLocation := L4
    };
    chooseConfirm() if currentLocation == L3 then {
      currentLocation := L5
    };
    !cancel(ref : RefID) if currentLocation == L4 && r == ref then {
      currentLocation := L0
    };
    !confirm(refOut : RefID) if currentLocation == L5 && r == ref then {
      currentLocation := L0
    }
  }
}
```

Listing 7.1: Simple Supplier action system

		deterministic model	model with internals
	precomputation	0.166	0.183
conformance check	mean	0.011	0.019
	median	0.007	0.012
	max	0.096	0.14
	min	~ 0	~ 0
	total	0.885	2.466
	# mutants	82	134

Table 7.1: Runtimes of operations involved in the test case generation for the supplier model. All durations are given in seconds.

7.1.2 Results

Experiments have been performed for measuring the runtime necessary to generate test cases from two versions of the supplier model. One deterministic model version does not involve internal actions, but rather combines the internal action `chooseCancel` with the output `cancel` and the internal action `chooseConfirm` with output `confirm`. The other model version with internal actions is shown in Listing 7.1.

The results of the runtime measurements as well as the number of mutants generated from each of the models are given in Table 7.1. The maximum search depth has been set to 20 for these measurements and the default strategy for equivalence checking has been used. Although the model is simple, the measurement results show that the introduction of internal action has a negative impact on runtime. They further show that the symbolic approach is able to efficiently handle large parameter spaces. The action `gq` for instance has a potential parameter space of size $2 * 10^{13}$ (without considering constraints in the guard). A concrete conformance checker, which enumerates all possible traces, would most likely fail to check conformance in reasonable time, thus the symbolic handling of parameters pays off because the computations (including precomputation) needed for test case generation only take about 3 seconds for the non-deterministic model.

7.2 Particle Counter

The comparison between the **sioco**-based and the **ioco**-based conformance was also presented at the USE-workshop [12]. However, the runtime measurement results differ from those presented before because the measurement setup changed. In order to reduce the runtime needed for processing other models, the conformance check implementation was changed and a newer version of Z3 is used in the current implementation. Unfortunately, these changes slightly increased the average runtime needed for the particle counter.

The particle counter is a system used by AVL, an industrial partner of the TRUCONF project[2]. As noted above, this use case has already been discussed and used in experiments in other publications such as Jöbstl's dissertation [58]. It is a device for measuring particle counts in exhaust gas and is used in automotive testbeds of AVL. The model of the device specifies the intended behaviour of the control logic, but does not deal with measurements in great detail.

The model needs to offer actions for:

- choosing different measurement modes, which determine whether accumulative or current particle concentration is measured,

[2] `http://truconf.ist.tugraz.at/` (last visit: 23.11.2015)

- setting a ratio, which determines the amount of particle free air, that is mixed with exhaust gas,
- performing a calibration
- and for performing maintenance tasks such as a leakage test or a response check.

For the action system model, the actions had to be divided into input and output actions. Input actions correspond to commands, which can be issued via the user interface. Output actions may for instance signal changes of the operating or system state as a result of command execution. Furthermore, outputs may also signal that commands were rejected.

The particle counter is in one of eight different operating states, may communicate either in *manual* or in *remote* mode and may be *ready* or *busy*. A command may be rejected if

- the particle counter is *busy*,
- the operating mode does not allow the execution of the command,
- or if the particle counter is in the wrong communication mode.

7.2.1 Specification and Modelling

As a result from previous work [58], there already existed two models of the particle counter, one is expressed via UML and one is expressed using action systems. However, it was not possible to use the existing model without any modification. Since the action system language used before is more complex than the language used in this thesis, it was necessary to translate the original action system into the new simpler language.

More concretely, the actions of the original action system model consist of several guarded commands, rather than of one guarded command. Furthermore, an action in the old model defines an additional guard, which affects all corresponding guarded commands. Due to the syntactic structure, the guarded commands belonging to an action will also be referred to as nested guarded commands, because they are enclosed in an action.

This modelling style is disallowed by the second well-definedness condition given for action systems in Section 2.3.2. A possible solution to this problem would be to discard this condition and allow for multiple guarded commands defining an action's behaviour. Since this extension of the conformance check would require several changes, it was not implemented for the following experiments. It will, however, be discussed in Section 8.1.1. For the remainder of this chapter, the focus will remain on action systems and the corresponding conformance check as presented in previous chapters.

Hence, a simpler approach involving translation is followed to generate test cases for the particle counter. For each action act with label a, guard g and n guarded commands with corresponding guards $g_1, g_2, \ldots, g_n$ and state updates $up_1, up_2, \ldots, up_n$, n actions are created, with labels $a_1, a_2, \ldots, a_n$, guards $g \wedge g_1, g \wedge g_2, \ldots, g \wedge g_n$ and state updates $up_1, up_2, \ldots, up_n$. Additionally, the newly created actions are defined to have the same parameters and action type as act.

Altogether, 69 input and 20 output actions have been created through this translation. The state of the new action system model is comprised of 10 variables and symbolic values are introduced through 26 parameter variables. Hence, the majority of actions does not define parameters. Nevertheless, symbolic execution may be more efficient than concrete execution since the enumeration of all parameter instantiations is time-consuming.

7.2.2 Results and Comparison

The measurement results for the conformance check are given in Table 7.2. Its first two columns shall be utilised for a comparison between the concrete **ioco** checker and the symbolic **ioco** checker. For this purpose, the maximum search depth was set to 20 for both applications and the and the default

		sioco	**ioco**	**sioco** (refactored model)
	precomputation	179.45	0	183.63
conformance check	mean	0.51	34.97	1.32
	median	0.02	3.44	0.01
	max	20.94	$2.96h$	17.4
	min	~ 0	0.03	~ 0
	total	936.64	$6.94h$	3051.74
	# mutants	1846	714	2318

Table 7.2: The execution times for the conformance check of the original model of the **sioco**-based and the **ioco**-based implementation are given in the first two columns. The execution times of the **sioco**-based conformance check of the refactored model are given in the third column. All durations are given in seconds, unless otherwise noted.

strategy for equivalence checking was used. The overall computation time needed by the symbolic conformance check was about $937s$ for 1846 mutants, while the concrete conformance check needed $6.94h$ for 714 mutants. It should be noted though that the concrete conformance checker is applied for test case generation from the original model defined using the more complex action system language with nested guarded commands. Although similar mutation operators have been used, the different syntactic structure induces a large difference in the number of mutants. Furthermore, it should be noted that the symbolic conformance check needs a precomputation time of about $180s$, whereas the concrete check does not perform any precomputation.

It can be seen that on average, the symbolic approach is about 68.6 times as fast as the concrete approach. This can be explained by looking at the mutant which caused the maximum runtime of the concrete conformance checker. The mutated action of this mutant has a potentially large parameter space, which is significantly decreased through the action's guard in the unmutated version of the action. The mutant, however, loosens this restriction by setting the guard to $True$ and as a result, the conformance check needs to enumerate all parameter combinations at each search step. Since the mutated action is an input, the mutation does not cause non-conformance which necessitates a complete exploration of the product graph. Hence, the concrete approach suffers from state space explosion or rather parameter space explosion, which is not an issue for the symbolic approach. Conforming mutants are also called equivalent and generally cause a large portion of runtime [4].

Furthermore, the ratio of the concrete median to the symbolic median is even larger than the ratio of the concrete mean to the symbolic mean. On the one hand this means that the gap between the concrete and the symbolic approach is even larger when long-running statistical outliers are excluded from the comparison. Thus for usage in practice, an implementation of a timeout should be considered for the symbolic approach, as this would result in exceptionally fast test case generation. If for instance, the timeout would be set to be $0.05s$, which is slightly larger than the median, the computation would take less than $92.3s$. In the worst-case, the conformance check of 924 of all 1427 non-conforming mutants would result in a timeout such that test cases would be generated from 549 mutants. As a result, the test suite generated about ten times as fast would still cover at least 37 per cent of the faults covered by the complete test suite.

On the other hand, the larger relative difference between median and mean of the symbolic conformance check suggests that the computation time of the symbolic check is heavier influenced by statistical outliers than the concrete check. The reason is that the median is not as influenced by outliers as the mean value, thus a large difference may indicate that there exists such an influence. A possible explanation is that the concrete conformance check, as indicated above, is usually slow for conforming mutants. This is an issue, which has been observed before in the context of concrete **ioco** checking [4]. While there are exceptions like the mutant corresponding to the maximum runtime of the concrete checker, the actual type of mutation has little influence on the runtime in many cases. This is not the case for the symbolic

conformance check because it uses precomputed data until the mutation has been executed. As a result, the depth at which a mutation is first executed influences the runtime. Hence, a conforming mutant may be checked for conformance very fast if its mutations is executed at a large depth with only a few search steps remaining. This leads to the conclusion that the runtime of the symbolic check can be harder to predict. However, this statement is not a general fact about symbolic conformance checks since the variation of runtimes is due to optimisations of the **sioco** check.

Now that the runtime of the **sioco** check has been discussed and compared to the runtime of the **ioco** check, it remains to be shown that the comparison is fair and does indeed make sense. While the same system is modelled, two different modelling styles have been used. The fact that the modelling style differs for the symbolic and the concrete conformance check suggests that the performance increase may at least partially be caused by the specific style used for the symbolic check. An argument reinforcing this suspicion is that assigning unique identifiers to guarded commands in addition to the action names adds information and thereby facilitates computation.

Furthermore, consider a situation during the symbolic conformance check, where the observation of an output o_1 is expected. This output o_1 corresponds to the output o in the original action system with nested guarded commands. Another output o_2 would correspond to the same output o in the original action system. However, a mutant producing the output o_2 would be detected to be non-conforming and the **sioco** check would be stopped, while in the original action system both outputs o_1 and o_2 correspond to the same label o. As a result, non-conformance would not be detected by the concrete approach and the search for conformance violations would be continued. Hence, adding unique identifiers may lead to a larger percentage of non-conforming mutants and thereby to a decrease in runtime.

However, while the modelling style may influence computation time in general, it is not an issue for the comparison considering the particle counter model. Firstly, because the nested guarded commands corresponding to an action can be chosen deterministically in the unmutated model. In other words, the guards are mutually exclusive, as there are no situations in which multiple guards of a single action are enabled. Consequently, it is unlikely that a mutation could cause a mutant to issue an output with a wrong identifier during the **sioco** check. The fact that the nested guards of an action are mutually exclusive also implies that adding unique identifiers does not introduce new information.

Secondly, an investigation of the number of conforming mutants and necessary search depths for detecting non-conformance permits the conclusion that the measurements performed with the symbolic and the concrete approach may be compared. As mentioned above, Aichernig et al. noted that conformance checking of conforming mutants causes a large portion of the overall computation time in model-based mutation testing [4]. Hence, the number of conforming mutants needs to be considered for both approaches. Due to the fact that similar mutation operators have been used, it is about the same in both cases, with about 22% for the symbolic approach and about 15% for the concrete approach. Given these numbers, it could actually be assumed that the symbolic approach is slower.

In addition to the proportion of equivalent mutants, the computation time generally rises with the search depth needed for detecting non-conformance. For this purpose, the search depth was continuously increased for each mutant until non-conformance was detected or the maximum search depth was reached. The necessary search depth values were stored. From these values, the median depth and the mean depth have been calculated for both approaches. Even if only non-conforming mutants are considered, the mean and median depth are higher for the symbolic approach. More specifically, the mean depth necessary to detect non-conformance is 10.2 for the symbolic approach and 8.3 for the concrete approach, while the median depth values are 10 and 8 respectively. Thus, the **sioco** check could be assumed to be slower based on necessary search depth as well.

Nevertheless, the different modelling style still implicitly modifies the interface of the system model, while the original action system faithfully models the system's interface. Hence, there may be event sequences, which correspond to indistinguishable action sequences in the interpretation of the original model, but which correspond to distinguishable action sequences in the interpretation of the new model because of action renaming. While this fact does not significantly affect the conformance check of

the particle counter according to the depth measurements, it needs to be considered by the test driver. A test driver must remove the identifiers and use the original event names known to the system for communication.

7.2.3 Refactored Model

An inspection of the particle counter model revealed that it contains actions, which are structured similarly to other actions. Consequently, a refactored model was created by combining actions with similar structure. The refactoring process did not follow a strict procedure like the action system translation, but rather focused on reducing the number of actions at the expense of adding parameters to the combined actions. Hence, it shall be illustrated through Example 7.1, which describes two steps in the process.

The refactoring reduced the number of output actions from 20 to 3 and the number of input actions from 69 to 23 at the expense of introducing new parameter variables. More specifically, the number of parameter variables was increased from 26 to 40. Although this is not a large increase in absolute numbers, it is large in relation to the number of actions.

Example 7.1 (Refactoring the Particle Counter).
In order to understand the following snippets the purpose of the variables `obs1`,`obs2` *and* `obs3` *needs to be explained. These variables are of the integer range type* `OutputEvent` *and model a FIFO-Queue of size* 3*. Whenever an input command causes one or multiple state changes, the corresponding output events are encoded as integers and enqueued. After that, an output action corresponding to the first event in the Queue signals the state changes and dequeues the event. Two such actions are given below.*

```
1  !spau_state1(ptime : Time)
2    if obs1 == 5 && ptime == 0 then
3  {
4    obs1 := obs2;
5    obs2 := obs3;
6    obs3 := 0;
7  };
8  !stby_state1(ptime : Time)
9    if obs1 == 7 && ptime == 0 then
10 {
11   obs1 := obs2;
12   obs2 := obs3;
13   obs3 := 0;
14 };
```

Listing 7.2: Two output actions in the original particle counter model

Since the structure of the actions is similar, they can be combined as shown in the following listing.

```
1  !dequeObs(event : OutputEvent,ptime : Time)
2    if ptime == 0 && obs1 == event && (obs1 == 5 || obs1 == 7) then
3  {
4    obs1 := obs2;
5    obs2 := obs3;
6    obs3 := 0
7  };
```

Listing 7.3: An action combining two output actions of the original particle counter model

While it is not necessary in principle to add a new parameter `event`*, it is added to distinguish the instantiations of the action* `dequeObs` *for the observation* `5` *and* `7` *respectively, as they correspond to different events of the system's interface.*

In order to discuss a situation, where it is necessary to add an additional parameter for conditional state updates another two snippets are given. Again, the first snippet defines a part of the original model and the second defines a part of the refactored model.

```
?startMeasurement3()
  if obs1 == 0 && !(aState == 1) && !(aState == 9) && busy == False && manual == False then
{
  obs1 := 1;
  obs2 := 0;
  obs3 := 0;
};
?startMeasurement4()
  if obs1 == 0 && busy == True && manual == False then
{
  obs1 := 1;
  obs2 := 0;
  obs3 := 0;
  readyIn := 30;
};
```

Listing 7.4: Two input actions in the original particle counter model

Both actions share some enabling conditions and have the same effect beside updating `readyIn` *differently, thereby it is possible to combine both as follows by introducing a new parameter for the post-state of* `readyIn`*:*

```
 ?startMeasurementCombined(nextReadyIn : Time)
  if obs1 == 0 && manual == False && (
    (!(aState == 1) && !(aState == 9) && busy == False && nextReadyIn == readyIn)
    || (busy == True && nextReadyIn == 30)
    ) then
{
  obs1 := 1;
  obs2 := 0;
  obs3 := 0;
  readyIn := nextReadyIn
};
```

Listing 7.5: An action combining two input actions of the original particle counter model

The new action does the same as the two actions `startMeasurement3` *and* `startMeasurement4`*, but it has a different interface. Although it seems as if the refactoring step removed the unique identifiers* `3` *and* `4`*, it may still be possible to infer which of the original actions would have been executed based on the parameter value of an execution of* `startMeasurementCombined`*. If the parameter is set to anything but* `30`*,* `startMeasurement3` *would have been executed.*

More problematic is that the parameter `nextReadyIn` *makes a part of the state visible which is actually ignored in the context of* ***ioco****. As a result, the modelling style used for the refactored model leads to a stricter conformance check, since the check takes parts of the post-state of an action into account.*

The reasoning behind this approach was that the symbolic conformance check is able to handle parameters very well, because they are treated symbolically and not enumerated explicitly, thus adding new parameters should not induce a large overhead in runtime. On the other hand, the path explosion problem is a prominent problem in the area of symbolic execution [32], therefore a reduction in the number of action should help reducing runtime, as it reduces the number of paths which need to be explored.

Furthermore, the previously mentioned problem concerning the implicit modification of the interface of the system is mitigated to some extend. A manual inspection suggests that the new reduced interface better reflects the system's interface as derived from the requirements. However, a closer analysis reveals that the increased number of parameters actually causes the system to offer a more fine-grained interface, which makes parts of the system state visible. In conclusion, the refactored version may have advantages regarding manual comprehension of the model, its LTS-interpretation still has about the same number of distinct paths as the model before refactoring. As mentioned in Example 7.1, this influences the **sioco**

check as well. Since the parameters of an action are considered to be a part of its label, the **sioco** check is sensitive to state changes if they are made visible through parameters.

Results

The measurement results are given in the third column of Table 7.2, whereby the default strategy was used again for equivalence checking. The overall computation time needed by the symbolic conformance check was $3051.74s$ for 2318 mutants. Although it was assumed that the time necessary to generate test cases would drop, the precomputation time of $183.63s$ is about the same as for the original model with unique identifiers. Even more surprising is the fact that the mean duration of the conformance check of the refactored model is larger than the corresponding value of the original model. A simple interpretation would be that the conformance check is not able to handle parameters very well. However, a closer investigation gives further insights.

Firstly, it shows that the computation time is spent in different parts of the application. For this purpose, the tool VisualVM[3] and more precisely the profiling and sampling capabilities of it were employed. This revealed that compared to the runtime for the original model, a larger portion of the runtime is spent for satisfiability checking. During the conformance check of the original model, the satisfiability checking time contributes 45.1 % to the overall computation time, while this portion rises to about 65.5 % for the refactored model. It should be noted though that simplification of formulas, which is also performed by Z3 and in turn decreases the time of satisfiability checks, makes up for 24.1 % of the conformance checking time of the first-mentioned model and 13.4 % for the refactored model. Nevertheless, this can be contributed to the larger number of different paths and thereby to the modelling style.

Secondly, a comparison between the original and the refactored model with regard to the number of actions shows that the number of actions significantly dropped, especially the number of output actions, as this number was decreased from 20 to 3. Since a larger number of actions allows for a more fine-grained syntactic mutation analysis, this implies that the reduced performance can be attributed to less effective optimisations.

However, like from the supplier model, it can be concluded that the symbolic conformance checker is able to efficiently handle large parameter spaces. This is possible, because an explicit enumeration of all parameters is not necessary. The action system contains for instance one action with five parameters of finite types spanning a parameter space of size $1,518,750$, while only a small portion of the parameters actually satisfies the guard of the corresponding action. An enumerative conformance checker such as Ulysses [3, 4] would most probably not be able to check such models unless it employs optimisations targeted towards situations in which only a small percentage of the parameter space needs to be considered.

7.3 Car Alarm System

This section will discuss the specification and conformance checking of different models of a car alarm system. Originally, it was provided by Ford as a use case within the EU FP7 project MOGENTES[4]. Since then, it has served as a benchmark for various mutation-based approaches to test case generation: for instance for model-based mutation testing based on refinement [9], based on **ioco** conformance [4] and based on timed automata [10].

[3] https://visualvm.java.net/ (last visit: 04.10.2015): VisualVM is a trouble-shooting tool for the Java Virtual Machine (JVM) and can for instance be used for profiling and sampling of Java-based applications with respect to memory consumption and runtime.

[4] http://www.mogentes.eu (last visit: 6.10.2015)

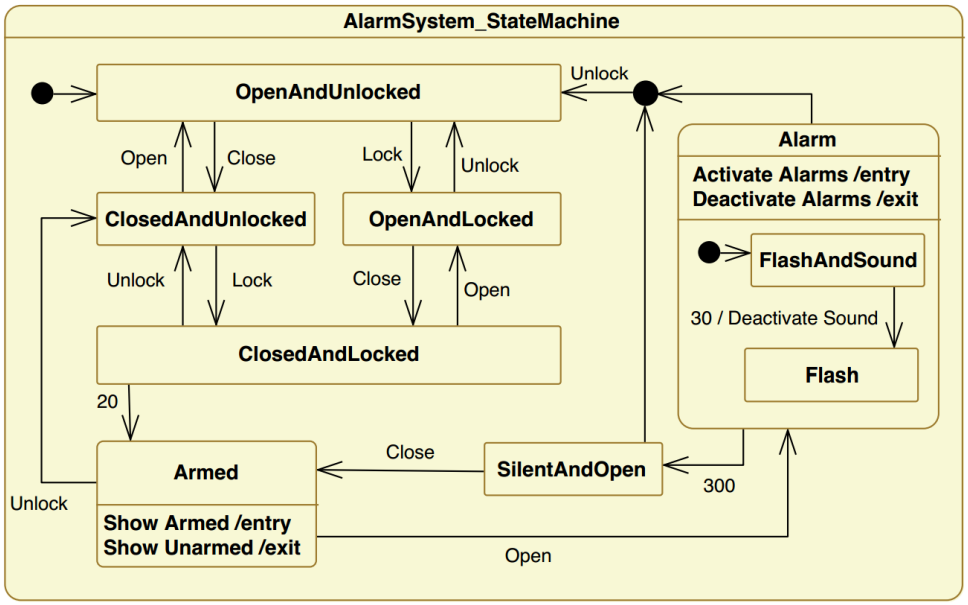


Figure 7.2: The car alarm system modelled via UML. This model has for instance also been used in [4, 9].

7.3.1 Specification and Modelling

It is nicely suited for evaluating these different approaches, as it involves important features of reactive systems, yet its specification is concise. More concretely, the following three requirements have to be satisfied by the systems:

R1 Armed. The car alarm system has to be armed 20 seconds after locking and closing all doors including the bonnet and the luggage compartment.

R2 Alarm. An alarm has to be activated, if one of the doors is opened by an unauthorised person while the system is armed. The alarm consists of an alarm sound and of hazard flasher lights. The sound has to be switched on for the first 30 seconds, while the lights have to be switched on for the first 5 minutes after activating the alarm. After the given periods of time, the respective alarm signals have to be turned off.

R3 Deactivation. The alarm can be deactivated at any point in time by unlocking the car. Deactivation is also possible, while the alarm is active.

A UML-model of the car alarm system is shown in Figure 7.2. The events, except those labelled with numbers, are transformed into input actions of an action system, while the effects are transformed into output actions. This means that the action system model contains for instance input actions for the event `Open` and an output action named `soundOff` for signalling that the sound is deactivated. An event labelled with a number n is a timed event, that is, the corresponding transition is fired after n time units have passed.

Two different approaches to translating these events will be used in the following. In the first approach, all actions will be associated with a time parameter, which specifies the number of time units that must pass, before the action can fire. Hence, the guard of the `armedOn`-action signalling that the `Armed`-state has been entered contains the constraint that the time since the last event must be equal to 20 seconds. The second approach follows the intuition that the passing of time is observable, thus an output action is added to the model, which signals that time has passed and increases the state of so-called clock variables. The time constraints are formulated using these variables. This approach is inspired by timed automata [64].

A closer investigation of the UML-model reveals that the car alarm system contains several loops, timing constraints and underspecification. The effect `Active Alarms` does not specify an order for the activation of flasher lights and sound for instance. However, its state space is relatively small unless continuous time is considered and stored via clock variables. As a result, one subsection will focus on determining whether the symbolic approach can efficiently handle time. Another challenge is the presence of loops, as these need to be detected through checking of symbolic state equivalence, which may be time-consuming.

Consequently, a concrete approach may be better suited, as the detection of loops is easier in the concrete case and because of the small state space. Nevertheless, the following experiments show that the answer to the question, whether the symbolic or the concrete approach should be used, needs to take the modelling style into account.

As before, it was not necessary to start modelling from scratch, but it was possible to build upon existing models. The system has already been modelled as an action system [9] and as timed automaton [10]. Additionally, Event-B models were provided by Severin Kann, who did his Bachelor's thesis under the supervision of Bernhard Aichernig. Each of these models will be translated in order to discuss different aspects of the application of the **sioco** conformance check.

7.3.2 Translation of Action System Model

Since the translation of action systems with nested guarded commands has drawbacks, other ways of translation should be considered as well. In the following, a procedure for translating such action systems will be given, which makes use of internal actions and creates action systems with the same interface and the same observable traces as the original system.

In order to understand the intuition behind the alternative approach to translation, consider the reason why a direct translation is not possible: nested guarded commands have different effects, that is, state updates are chosen depending on which of the guards are satisfiable. This is not possible in the action system language used in this thesis, because each action may only have exactly one body. However, it is possible to simulate the desired behaviour by executing an observable action under the same conditions as in the original action system, but delegating the state update to an internal action. The internal action should be executed under the same conditions as the corresponding nested guarded command in the original model.

Hence, the translated action system shall execute alternating sequences of observable actions and internal actions, which perform the state updates. In order to enforce that only those internal actions are executed, which correspond to guarded commands of the last executed observable action, the state needs to be extended by an additional variable storing the last observable action. Since nested guarded commands may define conditions involving parameters, the state needs to be further extended. More concretely, it is necessary to create one additional state variable for each parameter that occurs freely in nested guards. This allows to store them in observable actions and to read them in internal actions. Note that for simplicity, it is assumed that the original action system contains only observable actions. Nevertheless, the translation can easily be adapted to allow for internal actions, either by treating them like observable actions or by employing the first-mentioned translation, as they are not part of the interface anyway.

Algorithm 8 Procedure for translating action systems with nested guarded commands. The created simple action systems are observably equivalent to their respective original action system.

Input: Original Action System oas
Output: Translated Action System as

procedure TRANSLATEWITHINTERNALS
  Create new action system as initialised with same type definitions, state and initialisation as oas
  Create an enumeration sort $Event$ defining a constant $None$
  Add a state variable $lastEvent$ of type $Event$ and initialised to $None$ to as
  for all $a \in actions(oas)$ **do**
    Add constant $name(a)$ to $Event$
    $nestedDisjunction \leftarrow \bot$
    $parasInGuards \leftarrow \{\}$
    for all $g \in guardedCommands(a)$ **do**
      $nestedDisjunction \leftarrow nestedDisjunction \vee guard(g)$
      $parasInGuards \leftarrow parasInGuards \cup (\mathsf{free}(guard(g)) \cap para(a))$
    end for
    $observableBody \leftarrow \{lastEvent \mapsto name(a)\}$
    for all $p \in parasInGuards$ **do**
      Add new state variable x of same type as p to as
      $observableBody \leftarrow observableBody \cup \{x \mapsto p\}$
      for all $g \in guardedCommands(a)$ **do**
        Substitute x for p in $guard(g)$ and $body(g)$
      end for
    end for
    Create an action a' of same type as a with label $name(a)$
    $guard(a') \leftarrow guard(a) \wedge nestedDisjunction \wedge lastEvent = None$
    $body(a') \leftarrow observableBody$
    Add a' to as
    for all $g \in guardedCommands(a)$ **do**
      $newGuard = guard(g) \wedge lastEvent = name(a)$
      $newBody = body(g) \cup \{lastEvent \mapsto None\}$
      Create an internal action i with unique label
      $guard(i) \leftarrow newGuard$
      $body(i) \leftarrow newBody$
      Add i to as
    end for
  end for
end procedure

The translation procedure is outlined in Algorithm 8. For this purpose, actions with nested guards are assumed to be of the form given in Listing 7.6. The algorithm basically creates an "empty" action system as with the same types and state as the original action system oas including an additional variable of type $Event$ storing the last executed action (Lines 2 to 4).

Afterwards, each observable action a is processed independently whereby the following steps are executed (Lines 5 to 33):

- a new enumeration constant corresponding to the name of a is added to $Event$ (Line 6)
- a disjunction over all nested guards of a is formed and the parameters in those guards are collected (Lines 7 to 12)
- the body of a is extended such that parameters referenced in nested guards are assigned to state variables and the nested guards are adapted to reference these state variables (Lines 13 to 20)
- a new observable action a', which is executable whenever a would be executable, is created and added to as (Lines 21 to 24)
- for each nested guard, an internal action, which can only be executed after a', is created and added to as (Lines 25 to 32)

```
1  (?|!)act(p_1 : t_1,...,p_m : t_m) if G then {
2    if G_1 then {
3      x_1 := a_1;
4      ...
5      x_n := a_n;
6    }
7    ...
8    if G_k then {
9      x_1 := a_1;
10     ...
11     x_n := a_n;
12   }
13 }
```

Listing 7.6: Structure of actions with nested guards

Example 7.2 (Translation of an Action of the Original Car Alarm System).
This example shows two different translations of an action with nested guarded commands. The action to be translated is the output action `flashOn` *which signals that the hazard flasher lights are turned on. It is given via pseudo code below:*

```
1  !flashOn(wait_time : Time) if wait_time == 0 && isFlashOn == False then {
2    if aState == 1 && fromArmed == 2 then {
3      fromArmed := 3;
4      isFlashOn := True;
5    }
6    if aState == 1 && fromArmed == 3 then {
7      fromArmed := 4;
8      isFlashOn := True;
9    }
10 }
```

Listing 7.7: An action with nested guarded commands

The outer guard specifies that the system must immediately execute the action and that the action may only be executed if the flasher lights are switch off. Additionally, the nested guards check integers modelling the state of the car alarm system. Basically, both nested guarded commands turn on the flasher lights and change the system state.

Since the action contains two nested guarded commands, the translation technique described in Section 7.2.1 creates two actions `flashOn1` *and* `flashOn2` *which are shown below. Hence, it simply appends indexes to the labels.*

```
!flashOn1(wait_time : Time)
  if isFlashOn == False && wait_time == 0 && aState == 1 && fromArmed == 2 then
{
  fromArmed := 3;
  isFlashOn := True
};
!flashOn2(wait_time : Time)
  if isFlashOn == False && wait_time == 0 && aState == 1 && fromArmed == 3 then
{
  fromArmed := 4;
  isFlashOn := True
};
```

Listing 7.8: Translation of an action via appending of indexes to labels

It can be seen that the translation creates two observably different actions. To overcome this problem, Algorithm 8 creates internal actions to model the nested guards. A translation of the `flashOn`*-action via creating internal actions is shown below.*

```
!flashOn(wait_time : Time)
  if wait_time == 0 && lastEvent == None && isFlashOn == False &&
    ((aState == 1 && fromArmed == 2) ||
     (aState == 1 && fromArmed == 3)
    ) then
  {
    lastEvent := FlashOn;
  };
  flashOnUpdate1() if lastEvent == FlashOn && aState == 1 && fromArmed == 2 then
  {
    lastEvent := None;
    fromArmed := 3;
    isFlashOn := True
  };
  flashOnUpdate2() if lastEvent == FlashOn && aState == 1 && fromArmed == 3 then
  {
    lastEvent := None;
    fromArmed := 4;
    isFlashOn := True
  };
```

Listing 7.9: Translation of an action via creation of internal actions

An action system resulting from a translation via Algorithm 8 is obviously more complex than an action system resulting from the other translation method. However, it is observably equivalent to the original action system. This is achieved by delegating state updates to internal actions which may have arbitrary names. Constraints involving the state variable `lastEvent` *ensure that appropriate state updates are performed after observable actions.*

Results and Comparison

As for the particle counter, the results of measurements performed with the **sioco** checker will be compared to the results of measurements performed with the concrete **ioco** checker. For this purpose, two different translations of an action system with nested guarded commands have been created. The **sioco** checker has been used to generate test cases for the translations, while the **ioco** checker has been used to generate test cases for the original action system. Furthermore, the search was set to 20 for both test case generator and the default strategy for equivalence checking was used by the **sioco**-based approach.

The results are given in Table 7.3, where the first column lists the results for a model, which was translated as discussed in Section 7.2.1, and the second column lists the results for a model, which was translated as defined by Algorithm 8. In the following, the first-mentioned model will be referred to as model with indexed labels, while the latter will be referred to as model with internals. Example 7.2 demonstrates how one of the actions with nested guarded commands is translated via both methods. The

		sioco (with indexed labels)	**sioco** (with internals)	**ioco**
	precomputation	0.49	1.5	0
conformance check	mean	0.02	0.75	0.13
	median	0.012	0.17	0.06
	max	0.27	61.25	1.72
	min	~ 0	~ 0	0.02
	total	10.75	1201.8	32.41
	# mutants	515	1605	255

Table 7.3: Runtimes for the car alarm system models based on the original action system model: execution times for the **sioco**-based conformance check of two different translations of the original action system model are given in the first two columns. The execution times of the **ioco**-based conformance check of the original model are given in the third column. All durations are given in seconds, unless otherwise noted.

third column of Table 7.3 lists the results for the concrete **ioco** check performed between the original car alarm action system and its mutants.

The overall computation time was $10.75s$ for 515 mutants for the model with indexed labels and $1201.8s$ for 1605 mutants for the model with internals. As for the particle counter, a lower number of mutants was checked by the concrete conformance check. More specifically, 255 mutants have been checked in $32.41s$.

As can be seen in Table 7.3, the symbolic approach is again faster for the model with indexed labels than the concrete approach. The difference is not as drastic as for the particle counter though, on average it is only 6.5 times as fast as the concrete check. This can be explained by considering the state space. Since the state space of the car alarm action system is comparatively small, the symbolic approach cannot fully take effect.

Furthermore, the comparison is actually not entirely fair considering the number of conforming mutants. While the percentage of conforming mutants is only about 14.6% for the model with indexed labels, it is about 22% for the original model. The mean search depth needed to detect non-conformance, however, is about 5.16 for the original model and 5.6 for the model with indexed labels. If conforming mutants are considered as well, the mean search depth is increased to 9.55 for the original model and to 7.84 for the model with indexed labels. Hence, the search space which needs to be covered by the **sioco**-based conformance check is smaller than for the concrete **ioco** check. It can be concluded that the lower average runtime is partially caused by the different modelling style.

The measurement results for the model with internals indicate that this way of translation is not useful in practice, as the runtime increased significantly. The symbolic conformance check takes on average 37.5 times as long as for the model with indexed labels. Nevertheless, an investigation of the model gave further insights into the test case generation using the **sioco** check.

First of all, it served as a test for the conformance check itself. It has been used to check whether the conformance check produces spurious counterexamples if internal actions are involved. Spurious counterexamples are traces returned by the conformance check, which do not actually lead to conformance violations. The tests showed that the **sioco** check does not produce such counterexamples.

Furthermore, the investigation of the model with internals has shown that the combination of angelic completion and internal actions can lead to situations, in which the number of symbolic states of the mutant grows enormously with increasing search depth. Additionally, it has been observed that mutations of internal action guards may also slow down the conformance check. This is a result of the need for computing a τ-closure at each search step without utilising precomputed data.

Nevertheless, the high runtimes do not indicate that models containing internal actions take long to

process in general. The main reason for the poor performance is the large amount of internal actions. While the execution of a step during the conformance check takes at most three satisfiability checks in deterministic models, a τ-closure has to be computed additionally, when the model with internals is executed. Even if nested guarded commands are allowed, the number of satisfiability checks for executing a step is relatively small. Additionally, the nested guards of the concerned action need to be checked as well. For the computation of a τ-closure in the model with internals, however, all nested guards of all actions of the original model have to be checked for satisfiability. Hence, the translation of action systems through Algorithm 8 causes a loss of information, which significantly increases runtime.

7.3.3 Translation of Timed Automata Models

In the previously described experiments performed with the car alarm system, time was modelled in the same way as for the wheel loader case study discussed by Aichernig et al. [4]. This way of modelling adds one parameter to each action, which is equal to the time that has passed before the action has been executed.

However, another way of modelling time shall be investigated in the following. This way of modelling time is inspired by timed automata, which consider the passage of time as an observation [64]. The action system models presented in the following were created after discussions with Florian Lorber, who provided timed automata models as a basis. Timed automata shall be shortly discussed in order to understand the translation. A more thorough discussion of the topic has for instance been presented by Bengtsson and Yi [16].

Timed automata are finite automata extended with a finite set of clocks. They are used to model real-time systems. The clocks are modelled by real-valued variables, which are increased by the passage of time. Time is considered to pass in states, while transitions are considered to take zero time. A transition is either labelled with τ, denoting it to be a silent, non-observable transition, or with an action label. The set of action labels can further be partitioned into disjoint sets of inputs and outputs.

Furthermore, constraints over clocks restrict the set of executable traces of a timed automaton. These constraints are either associated with states of the automaton or with transitions functioning as guards. A transition's guard defines the clock states in which the transition may be taken. Clock constraints associated with states are also called invariants and limit the number of time units the system may stay in some state. Figure 7.3 shows a timed automaton model of the car alarm system. The model has also been used for test case generation by Aichernig et al. [10].

There exist various ways to check conformance of timed automata including language inclusion, bisimulation [16] and also a timed variant of **ioco** called Timed Input Output Conformance (**tioco**) [64]. In this context, runs of timed automata are usually defined to be alternating sequences of delays and transitions, which are taken along some trace [16]. Hence, an action system simulating a timed automaton needs to create actions corresponding to transitions and another output action *delay*, which simulates the passage of time. Additionally, it needs to ensure that in between two actions, the *delay*-action is executed. In order to conform with timed traces as defined by Aichernig et al. [10], the translation of timed automata into action systems ensures that the execution starts with a *delay*.

If several transitions are labelled with the same action, the corresponding labels in the action system need to be extended by adding unique identifiers as for the translation of nested guarded commands. The rest of the translation is actually similar to the translation of IOSTSs into action systems, thus a location data type corresponding to timed automata locations and a state variable $location$ storing the current location are introduced as well.

In addition to that, the state is further extended by real-valued clocks defined by the timed automaton that is being translated. The data type $Real$ has been used for this purpose. A clock reset is modelled via an assignment, which sets the corresponding clock to zero.

Finally, clock constraints need to be considered during the translation as well. Time guards on transitions are simply added to the guards of the corresponding actions. Since invariants constrain the

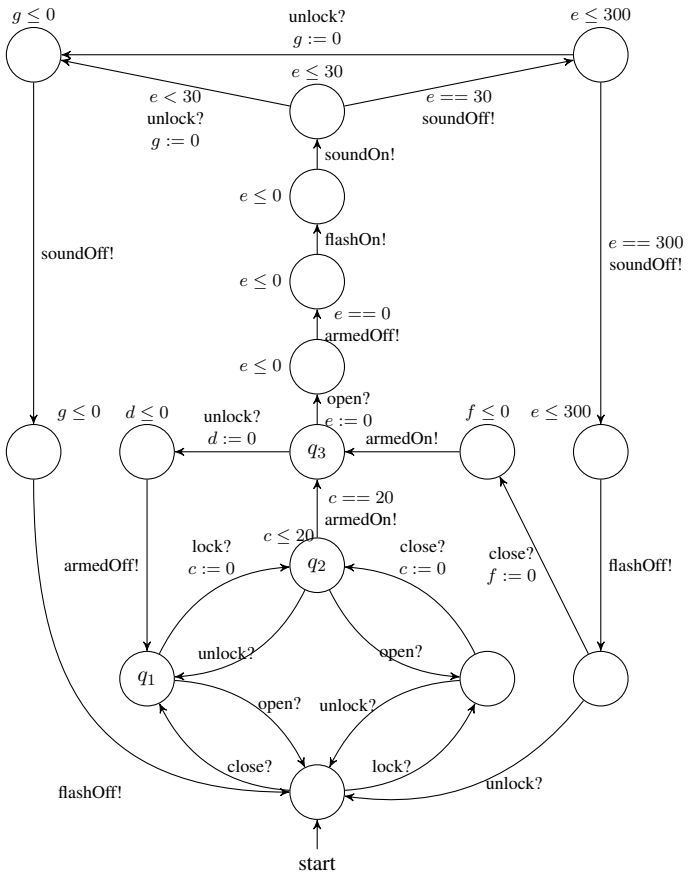


Figure 7.3: A timed automaton model of the car alarm system. The model has also been used by by Aichernig et al. [10].

waiting time in a state, the parameter of the *delay*-action, which corresponds to the waiting duration, needs to be constrained accordingly. It is necessary to define conditions depending on the current value of $location$ for this purpose. Hence, the guard of the *delay*-action is a conjunction of implications $location = l \rightarrow inv(l)[q'_c]$, where l is a constant corresponding to a location in the timed automaton, $inv(l)$ is the corresponding invariant and $q'_c = \bigcup_{c \in Clocks} \{c \mapsto c + duration\}$ is the post-state of the clocks after increasing their value. Additionally, it needs to be ensured that the invariant holds in the post-state of all other actions as well and the *delay*-duration should be restricted to non-negative values.

Example 7.3 (Translation of a timed automaton modelling a car alarm system).
In the following, the translation of the timed automaton shown in Figure 7.3 will be discussed. The automaton is translated into an action system $\mathcal{TAS}$ as described above. Since the timed automaton defines five clock variables c, d, e, f, g, the state of $\mathcal{TAS}$ is defined by the following listing:

```
1 state {
2   location : Location;
3   c : Real; d : Real; e : Real; f : Real; g : Real;
4   doDelay : Bool;
5 }
```

Listing 7.10: State definition of an action system modelling a real-time system

Line 2 defines a variable storing the current location of the corresponding timed automaton. The clock variables are defined as state variables of type $Real$ in Line 3. A Boolean variable `doDelay` *defined in Line 4 indicates whether the system should execute a* delay*-action or a discrete action in the next step.*

A discrete action modelling one of the `armedOn`*-transitions is shown below:*

```
1 !armedOn() if !doDelay && location == ClosedAndLocked && c == 20 then
2 {
3   doDelay := True;
4   location := Armed;
5 };
```

Listing 7.11: An action modelling a transition of a timed automaton

The action may only be executed if `doDelay` *is set to* `False`*, the system is in the location* `ClosedAndLocked` *and exactly* 20 *time units have passed since clock* `c` *had been reset. An execution of the action causes the system to change into the* `Armed`*-state and to execute the* delay*-action next.*

A part of the delay*-action is given in the listing below.*

```
1     !delay(duration : Real)
2     if doDelay && tics >= 0 &&
3     (!(location == ClosedAndLocked) || (c + duration <= 20)) &&
4      ...
5     (!(location == SilentAndUnlocked ) || ( g + duration <= 0)) then
6     {
7     c := c + duration; ... g := g + duration;
8     doInc := False;
9     };
```

Listing 7.12: An action modelling the passage of time

The delay*-action is an output signalling that time has passed. It may only be executed if* `doDelay` *is set to* `True` *and with non-negative parameter values. Furthermore, it defines conditions such as* `(!(location ==ClosedAndLocked)||(c +duration <=20))` *which can be read as "if the system is in location* `ClosedAndLocked`*, time may only pass as long as the value of clock* `c` *is smaller than or equal to* `20`*". An execution of* `delay` *increases the value of all clocks and causes the system to execute a discrete action in the next step.*

Problems and Inconsistencies. Essentially, all runs of a timed automaton can be executed by an action system, which is translated as described above. It should be noted though that appending indexes to labels, which is needed if there are multiple transitions with the same label, causes different observable traces.

Beside this difference, there is another one, which becomes apparent during the conformance check. Using timed automata, it is possible to define timeout durations after which an output is expected explicitly. In the **sioco** conformance relation, however, the notion of quiescence is introduced, which essentially signals a timeout of undefined duration. In other words, a system is considered to be quiescent if an undefined amount of time units passes without observing any output.

Hence, these two types of timeouts are checked in parallel during the conformance check, if the **sioco** checker is used without adaptations. In the following, a problematic situation shall be discussed, which arises from this mix of timeout definitions: consider a quiescent state, which is the same for mutant and specification and the specification executes an input action, which is mutated and thereby not enabled in the mutant. In the next specification state, a Boolean flag triggers that the *delay*-action is the only enabled action. The mutant, however, stays in the same state as before, because an angelic completion is performed for it. As a result, the mutant state is still quiescent, thus it waits for an undefined amount of time without producing any outputs. Since the specification may execute the *delay*-output, it is not quiescent.

Consequently, the mutant would be detected to be non-conforming. This is problematic, because a non-zero delay-duration chosen for the execution of the *delay*-action, would essentially correspond to a timeout of the specification, thus both action systems would wait and do nothing in the state reached after the execution of the input. A simple approach to counter this problem is to disable checks for quiescence.

However, this leads to the problem that a large number of mutants would be detected to be conforming, which would be considered non-conforming by a **tioco** conformance check. Consider the product state reached after executing the input action by the specification and after performing an angelic completion for the mutant: the mutant is quiescent, thus it does not produce any observation, while the specification can only produce the *delay*-observation. Since the product state reached after executing the delay would not be satisfiable, the conformance check would be stopped and the mutant would be considered to be conforming.

The angelic completion of a conformance checker based on **tioco** would need to take this problem into account and manipulate the flag, which signals that a delay has to be executed next. Furthermore, the mutation of the *delay*-action should only be performed in a way, such that the action is not disabled in all states. An action system with an unsatisfiable *delay*-guard would model a system, which does not do anything, as the translation requires the first action to be a *delay*.

In addition to the angelic completion, the handling of internal actions would need to be adapted as well. The reason for this is that consecutive delays, only separated by the execution of internal actions, would need to be combined into one delay [64].

Consequently, the conformance check was not adapted for the experiments with translated timed automata, because the main focus lies on the **sioco** conformance relation. The problem concerning quiescence shall rather be tackled on model-level and a solution on conformance check-level shall be discussed in Section 8.2.

A possible solution to the quiescence-problem would be to loosen the restriction that only alternating sequences of delays and actions may be executed. The *delay*-action should be allowed to be executed in all states. Furthermore, mutations should not disable the *delay*-action as discussed above. The second requirement can be fulfilled by restricting the mutations of the *delay*-guard, such that only the *increment*-operator is applied and only constants are incremented. This actually mimics *Change invariant*-mutation operator applied on timed automata [10].

Although this approach would avoid a mix of different notions of timeouts, as the models never become quiescent, it faces another problem. A similar problem, has been observed by Aichernig et al. [4], which led to the introduction of parameters for modelling time as has been done in Section 7.3.2. Since the *delay*-action, which is an output action, is enabled in all states, the system contains a large number of mixed states. A mixed state is a state in which both inputs and outputs are enabled. Such states are problematic for testing in general, as the tester may choose to send an input, while the SUT may produce an output as allowed by the model. Consequently, the tester would need to block the output from the SUT, which is unrealistic, as the communication between tester and SUT is often asynchronous [35].

Discussion of Measurements. In conclusion of the discussion above, test case generation based on either of the modelling strategies faces problems. Nevertheless, time measurements have been performed for the conformance check and the results are given in Table 7.4. For this purpose, the timed automaton given in Figure 7.3 was translated using both strategies, that is, with and without alternation of delays and actions. To illustrate the structure of action systems modelling real-time systems, Example 7.3 shows code snippets of the car alarm system model with alternating delays and actions.

The generated test cases are not meant to be executed though. The measurement results shall rather give an insight on whether symbolic execution is suited for conformance checking of models for real-time systems. For this purpose, the conformance checking time will be compared to the runtime required by a **tioco** conformance checker based on bounded model-checking, which has been developed by Aichernig et al. [10].

The measurement results for the bounded model-checking approach differ from the results given by Aichernig et al. [10], as the measurement setup changed. More concretely, the measurements have been performed on a different computer, the same computer that has been used for all other experiments discussed in this chapter. In addition to that, version 4.3.2 of Z3 has been used instead of version 4.0.

		sioco (alternating)	**sioco** (non-alternating)	**tioco**
	precomputation	32.97	102.82	0
conformance check	mean	3.16	3.77	0.85
	median	0.013	0.015	0.96
	max	29.1	93.59	4.65
	min	~ 0	~ 0	0.09
	total	4811.52	5329.06	1092.3
	# mutants	1522	1412	1285

Table 7.4: Runtimes for car alarm system model based on a timed automata model: the execution times for the **sioco**-based conformance check of two different translations of the original timed automaton are given in the first two columns. The execution times of the **tioco**-conformance check of the original timed automata model are given in the third column. All time values are given in seconds, unless otherwise noted.

The maximum search depth has been set to 12 for all conformance checks and the strategy *Quantifier Elimination After Timeout* has been used for checking state equivalence. It should be noted though that the observable depth in the context of **tioco** is defined differently than for action systems. One step of a timed trace consists of the execution of a transition together with a delay, thus the search depth for the model with alternating actions and delays has actually been set to 24 for the **sioco** conformance check. However, the search depth has been set to 12 for the other model, because it does not require that actions need to be interleaved with delays. Additionally, it was not possible to finish the conformance check with search depth 24 in reasonable time.

Considering the action system with alternating actions and delays, it can be seen that on average the bounded model-checking approach is more than three times as fast as the **sioco**-based approach. The median duration of the symbolic execution approach, however, is much lower than the median duration of the **tioco** check. In contrast to this, the difference between the median and the mean of the **tioco** check is low. Hence, the runtime seems to be more constant when bounded model-checking is performed. Since the median duration of the **sioco** check is very low, a timeout could be used to stop excessively long-running conformance checks as discussed before in Section 7.2.2.

Although conformance checks between the second action system and its mutants have been performed with only half the maximum search depth, they required longer computation times than the conformance checks of action systems with alternating delays and outputs. Since the precomputation runtime is also higher, the lower performance is most probably caused by the structure of the specification model and not by the structure of specific mutants. A possible explanation is that checking symbolic state equivalence is more difficult. As the delays need to be interleaved with actions for the execution of the first model, the clocks are frequently reset. Thus, the symbolic clock states contain fewer *delay*-parameters than the symbolic clock states of the second model. Hence, a lower number of variables is existentially quantified for state equivalence checks, which results in faster checks and overall better performance.

Measurements for Non-deterministic Models. In addition to the previously discussed models another action system model defining one internal action shall be examined as well. The internal action adds a margin of tolerance of two seconds to the twenty seconds, which the system waits before switching into the *Armed*-state. A corresponding timed automaton, also containing a silent transition, serves as a reference for the conformance checking runtime.

The presence of internal actions complicates the **tioco** conformance check, as the efficient conformance check presented by Aichernig et al. [10] is restricted to deterministic models and thus requires determinisation. An efficient procedure for determinising time automata containing silent transitions

		sioco (complete-depth 12)	**tioco** (partial 1-depth 8)	**tioco** (partial 2-depth 12)
	precomputation	565.33	0	0
conformance check	mean	0.4	49.94	2.16
	median	0.03	44.3	2.93
	max	28.6	558.32	989.7
	min	~ 0	0.674	0.1
	total	717	9987.4	2638.58
	# mutants	1768	200	1221

Table 7.5: Runtimes for non-deterministic real-time car alarm system model: the execution times for the **sioco**-based conformance check of the translation of the non-deterministic timed automata model are given in the first column. The execution times of the **tioco**-based conformance check of both partial non-deterministic models are given in the second and third column. All time values are given in seconds, unless otherwise noted.

has been developed by Lorber et al. [68]. This procedure has been performed prior to the conformance checks of the timed automaton and its mutants.

Table 7.5 lists the measurement results for the non-deterministic model containing one internal action. Note that the action system was translated from a timed automaton in a way such that all executions of discrete actions are followed by delays. The complete timed automata model could not be processed by the **tioco** conformance check, as the determinisation performed during preprocessing led to a state space explosion, which rendered the approach infeasible. Instead, the model was partitioned into two partial models. The first partial model describes the locking, unlocking, closing and opening of doors and the arming of the car alarm system, whereas the second partial model contains only one path to the *Armed*-state but covers the rest of the systems. These partial models have been created by Florian Lorber and are shown in Figure 7.4.

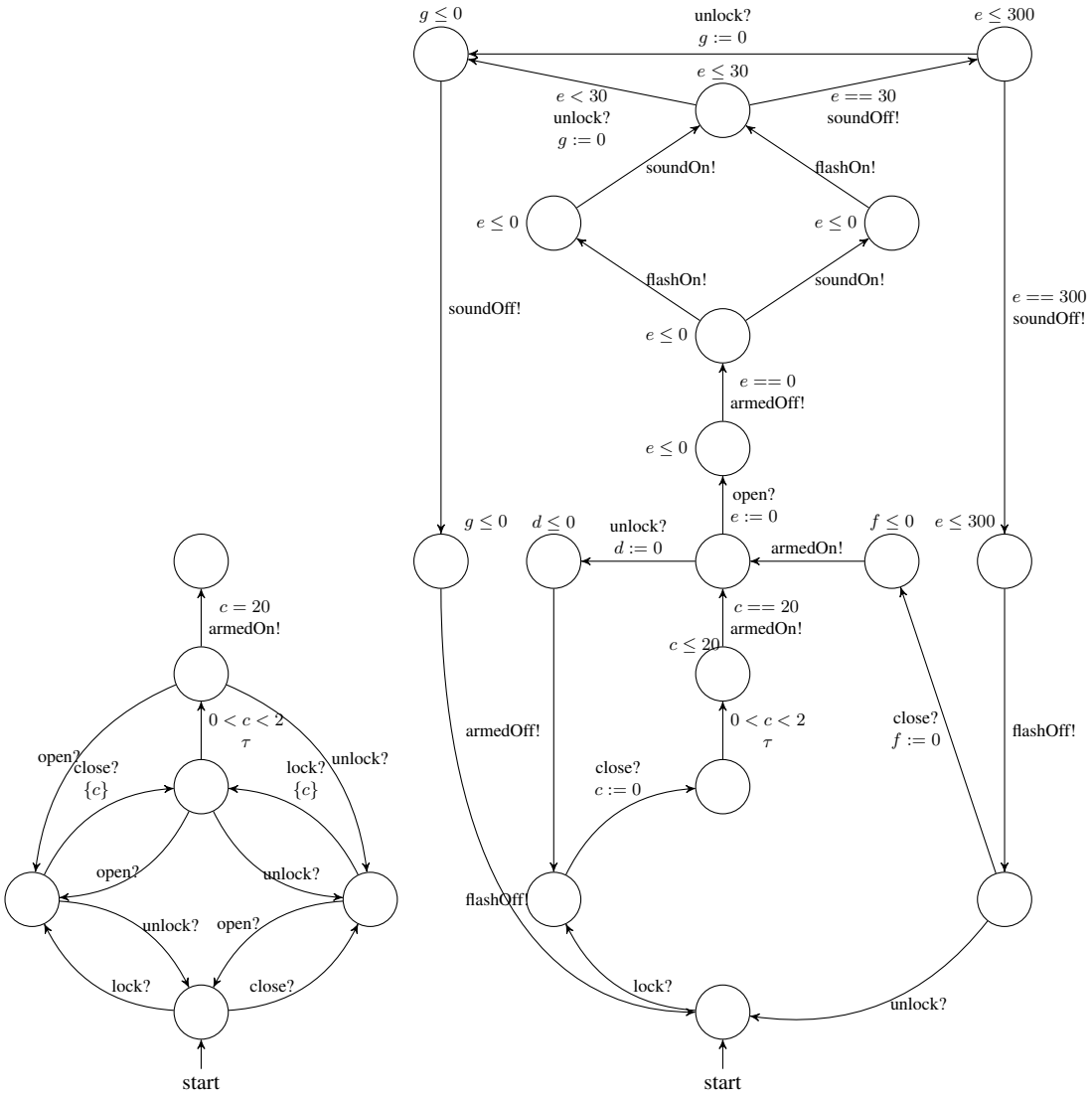
The symbolic execution-based conformance check on the other hand was able to process the complete model. Surprisingly, it has on average even been faster than both the **sioco** and the **tioco** conformance check of the deterministic model. This can be attributed to the fact that the introduction of a silent transition led to a high proportion of non-conforming mutants. Out of the 1768 tested mutants, 1766 were deemed to be non-conforming. It should be noted though that the precomputation took 565.33 seconds. By considering precomputation to be a part of the conformance check, the total runtime increases to 1282.33 seconds. Strictly speaking, the bounded-model checking approach involves precomputation as well as it is necessary to determinise timed automata prior to the conformance check. Since the combined runtime of the determinisation of all models is very low, it is not explicitly given.

In this context, it should also be mentioned that it was possible to efficiently process both partial models using symbolic execution as well. The runtimes are not given, however, as the combined runtime for both models was lower than that of the conformance check of the complete model.

Comparing the conformance checking runtimes of both approaches shows that symbolic execution is significantly faster. It is able to process the complete model in less time than the bounded-model checking approach can process either of the partial models. This is a result of the state-space explosion caused by determinisation.

In conclusion, these experiments suggest that symbolic execution may be an efficient approach to conformance checking of real-time system models. Since the median duration of the conformance check is generally low, many mutants can be processed very fast. Furthermore, it shows advantages when models containing internal actions are checked for conformance.

An actual symbolic check of **tioco** conformance may perform even better, as it would be possible to optimise the check with respect to time. On the other hand, it may also perform worse, because delays



(a) Partial model 1 **(b)** Partial model 2

Figure 7.4: Partial timed automata models of the car alarm system with a silent transition modelling a non-deterministic delay of entering the *Armed*-state.

interleaved with internal action executions need to be combined. A more thorough discussion of this topic will be given in Section 8.2. However, a non-symbolic conformance check is impossible because the models use reals, which cannot be enumerated.

7.3.4 Verification during Stepwise Development

This subsection will discuss the verification of **sioco** conformance between two arbitrary action system models, that is, the restriction to first-order mutations will be lifted for the following experiments. More concretely, the applicability of the **sioco** conformance checker for model-checking in stepwise development of test models will be investigated. Furthermore, factors influencing performance will be examined as well. The general approach and its usefulness have been discussed in Section 1.5 and in Section 5.2.

Hence, as noted in said sections, the **sioco** conformance of two models will be checked, where one model is more abstract and the other model is a refined version of the first. In this context, the concrete model will be considered to be an implementation of the abstract model, which serves as the specification.

Translation. As in the previous subsections, the experiments will be based on existing models of the car alarm system. These models have been defined in Event-B by Severin Kann as part of his Bachelor's thesis. Event-B is inspired by action systems [1] and thus the Event-B notation is very similar to action system language presented in this thesis. As a result, the translation of the Event-B models was comparatively simple because they do not use complex set-theoretic constructs, but only integers and sets defining constants. Hence, the unbounded integer data type provided by Z3 was used for all integral numbers and enumeration sorts have been defined for sets of constants. Furthermore, events have been transformed into almost identical actions. Since Event-B does not distinguish between input and output events, it was necessary to assign action types.

Modelling. The Event-B models are similar to the deterministic timed automata model in some aspects. Events in the models represent transitions between states, that is, there exist events for operations such as opening of doors and turning on the alarm. In addition to that, the passage of time is modelled via observations.

However, specific details concerning time are treated differently from the timed automata formalism: for instance, discrete time is assumed, that is, time is modelled with integers. A timer (clock) is set to some constant value and decreased rather than increased. Furthermore, two different events represent the passage of time. One event is fired if the delay triggers some other action and the other is fired if the delay has no effect.

Altogether, four different models have been created by Severin Kann. Additionally, he performed proofs to show that each model refines its predecessor if it has one. Each refinement step adds some information, outlined in the following:

Abstract Model: introduces the basic structure and functionality, but without considering time. Hence, the system may switch into the *Armed*-State at any point in time if the doors are closed and locked.

First Refinement: introduces events representing the passage of time and adds time-related constraints to existing events.

Second Refinement: constrains the passage of time such that each delay may only take exactly one time unit.

Third Refinement: while in the more abstract models, the system's state is represented by one variable, which takes constant values such as `open_unlocked`, this stage of refinement does not use these constants anymore. It rather models the system's state via a set of Boolean flags. One of these flags for instance encodes the information, whether the doors are open.

From these descriptions, it becomes apparent that a **sioco** conformance verification between the most abstract and the other models is not possible because the first refinement adds new output events representing the passage of time. Strictly speaking, a conformance check would be possible, but reveal conformance violations although the refinements model intended behaviour. Hence, conformance will only be checked between first and second, second and third, and first and third refinement.

Experiments. Intuitively, both the second refinement and the third refinement should be valid implementations of the first refinement with respect to **ioco**. The second refinement strengthens the guards of the actions representing delays, thus it produces less outputs than the first refinement, which is allowed by **ioco**. The third refinement merely changes the system state, while it is observably equivalent to the second refinement. Hence, it is a valid implementation as well, as **ioco** ignores the state of a system.

The conformance checks performed during the experiments confirmed this intuition. They were performed with a search depth large enough to conclude that each refinement is a valid implementation of its respective specification for all possible sets of traces. Stated differently, the experiments have shown that there does not exist any trace of any length, which would reveal non-conformance. Recall the

	$2^{nd}ref$ **sioco** $1^{st}ref$		$3^{rd}ref$ **sioco** $2^{st}ref$		$3^{rd}ref$ **sioco** $1^{nd}ref$	
runtime	*on-the-fly*	*optimised*	*on-the-fly*	*optimised*	*on-the-fly*	*optimised*
precomputation	0	1.375	0	5.242	0	1.33
conformance check	115.59	2.629	8.154	1.079	116.696	2.475
total	115.59	4.004	8.154	6.321	116.696	3.805

Table 7.6: The runtimes of the conformance checks between refinements of the car alarm system. All durations are given in seconds.

definition of sufficiency of exploration depth for the product graph given in Section 5.2: if the conformance checker stops searching before hitting the maximum exploration depth for all traces, unbounded **ioco**-conformance between implementation and specification can be concluded.

In order to be able to avoid excessively large search depths, the durations needed for triggering events in the car alarm system have been reduced. As a result, a search depth of 22 was found to be the lowest depth, which satisfies the mentioned sufficiency condition for the conformance checks between all possible combinations of refinements. Hence, the runtime measurements have been performed with this search depth. Their results are presented in Table 7.6. They were carried out for two different configurations: in the first configuration, *on-the-fly*, all optimisations have been disabled, while in the second configuration, *optimised*, only those based on first-order mutants have been disabled. Thus, the conformance check may still profit from a precomputed symbolic execution graph and from precomputed symbolic state equivalence classes in the second configuration. Additionally, the strategy for checking state equivalence has been set to *Quantifier Elimination after Timeout*.

The measurements demonstrate that precomputation pays off even if only one conformance check is performed, because the total runtime of the *optimised* strategy is lower for each of the three checks. The main causes for this behaviour are: (1) it is possible to apply the approximated equivalence check of product states, which needs the precomputed symbolic state equivalence classes. Furthermore, (2) as the symbolic execution graph contains loops, it encodes information about executable traces, which may be longer than the actual traces in the graph. This reduces the total number of satisfiability checks necessary for the execution of the specification.

The experiments show that the **sioco** conformance check may effectively be used for model-checking during stepwise development. Note that concrete approaches are likely to fail for the conformance checks involving the first refinement because unbounded data types are used. The conformance check between second and third refinement, however, is amenable to a concrete approach, because all time steps are restricted to one time unit.

Beside the considerations whether a concrete or symbolic approach should be used, another issue needs to be considered for checking **ioco** conformance during stepwise development. It has been pointed out in Section 4.4 that performing angelic completion can be seen as a mutation and the angelic completion of a model may not conform to the original model. Nevertheless, it is essential for checking conformance between refinements, as input-enabledness cannot be assumed in general. Additionally, Tretmans noted that **ioco** is reflexive for input-enabled models, but comparing non-input-enabled models does not make sense because **ioco** is not defined for those [79].

Hence, the angelic completion of a refinement of a model M may not conform to M, despite solely changing the representation of the system's state or internal actions. This is especially likely if the angelic completion of M does not conform to M. A possible solution to this problem would be a manual inspection to determine whether witnesses of non-conformance actually lead to undesired behaviour. If they do not, they could be excluded from the product graph and the conformance check could be performed again. Another solution would be to be disable angelic completion for this type of verification task. However, such an approach would not verify **ioco** conformance because **ioco** requires input-enabled implementations.

7.4 Models using Complex Data Types

This section will deal with complex data types and associated problems. In this thesis, complex data types are considered to be data types parameterised with other data types. As noted in Section 6.1, the conformance checker provides maps, sets and records. The following discussion is based on experiments with two small examples involving all of these data types. Although the conformance check is able to process these models in principle, it faces severe efficiency problems for large search depths. Hence, this section will focus on these issues and discuss how they may be overcome.

7.4.1 Models

Although both of the models below are small, they define only three and four actions respectively, they demonstrate the mentioned efficiency problems. As a result, larger models will not be investigated.

Set-Buffer. The first model specifies the behaviour of a simple buffer and defines one state variable of set data type. Experiments have been performed by either setting the corresponding element data type to an enumeration type or to an integer range type. The buffer defines one input action and three output actions. Elements can be inserted via the input action. These elements may be selected and shown to an observer by one output action. After presenting the element to the observer, the elements are removed from the buffer. The other two outputs signal whether the buffer is empty and whether the buffer contains a specific constantly specified element.

Tuple-Map. The second model defines a state variable of map data type, which maps from unique integer identifiers to pairs of integers. The pairs are modelled via records and can be inserted via an input action. Another input enables the user to specify the identifier of a pair, which should be presented to the user through an output of the system.

7.4.2 Experiments

Experiments soon revealed that checking equivalence of symbolic states is difficult and cannot be performed by Z3 if states at depths larger than two are involved. More concretely, Z3 is not able to eliminate quantifiers in the equivalence conditions derived for these models. This can be attributed to the fact that the underlying sort of the set- and map-data types is the array-sort provided by Z3, which has not been used in previous experiments.

As a result, most of the equivalence checks failed and the performance overhead induced by the failed attempts to check satisfiability was significantly larger than the performance increase stemming from the succeeded attempts. Consequently, equivalence checks for all types of symbolic states have been disabled for the following experiments. Since the equivalence checks are actually used to tackle the path explosion problem seen in symbolic execution, the performance of the **sioco** conformance check suffers from this problem when complex data types are used.

The path explosion problem results from the growth of execution paths, which is exponential in the number of branches [32]. Consequently, the number of observable traces of action systems grows exponentially with increasing search depth. To illustrate this issue, conformance checks have been performed with varying maximal search depth. The corresponding measurement results include runtime data and the size of the symbolic execution tree. They are given in Table 7.7 for the set-buffer and in Table 7.8 for the tuple-map.

For these measurements, an integer range data type was chosen as element data type of the set-buffer. Furthermore, standard mutation operators as well as a set-specific mutation operator, which removes the addition of a new element, were used, resulting in 28 and 30 mutants for set-buffer and the tuple-map respectively.

			conformance check				
depth	size of tree	precomputation	mean	median	max	min	total
2	14	0.03	0.008	0.004	0.05	~ 0	0.22
4	175	0.18	0.03	0.003	0.35	~ 0	0.77
6	2353	2.16	0.18	0.002	2.639	~ 0	4.94
8	33098	107.99	1.36	0.002	20.288	~ 0	38.18
10	478192	$7.5h$	32.48	0.002	664.37	~ 0	909.462

Table 7.7: Runtimes for different steps of the test case generation for the *set-buffer*. Additionally, the size of the symbolic execution tree is given in terms of states. All durations are given in seconds, unless otherwise noted.

			conformance check				
depth	size of tree	precomputation	mean	median	max	min	total
3	16	0.04	0.017	0.012	0.06	~ 0	0.5
6	164	0.31	0.06	0.02	0.58	~ 0	1.87
9	1631	2.91	0.53	0.03	7.53	~ 0	15.36
12	16168	45.17	8.08	0.04	116.48	~ 0	234.4
15	160219	4183.43	237.09	0.1	3943.48	~ 0	6875.49

Table 7.8: Runtimes for different steps of the test case generation for the *tuple-map*. Additionally, the size of the symbolic execution tree is given in terms of states. All durations are given in seconds, unless otherwise noted.

It can be seen that the total runtime increases dramatically, as the search depth is increased. This is caused by (1) an exponential growth of the symbolic execution tree, and (2) by the increasing complexity of satisfiability checks of path conditions at larger depths. Hence, it can be concluded that the path explosion problem needs to be overcome by means other than equivalence checks in order to efficiently generate tests from models with complex data types. The importance of this measure is aggravated by the fact that the tested models are very small with only four and three actions respectively, as compared to the 89 actions defined by the particle counter, an industrial use case.

The fact that the median conformance checking runtime stays approximately the same for all set-buffer measurements and does not grow as fast as the average runtime for the tuple-map can be explained based on the mutants. Since the models are very simple, a large number of mutants is detected to be non-conforming in the initial state. As a result, the growth of the number of traces does not influence the conformance checking runtimes for these mutants. It follows that the median duration does not change significantly as well.

In the description of the set-buffer, it has been noted that the element data type was set to be an integer type or an enumeration type. If the set is parameterised with an enumeration type, Z3 is actually able to eliminate quantifiers, but at the expense of creating large formulas. This increase in size as compared to the original quantified formulas leads to an increase of the computation time necessary to check symbolic state equivalence.

Additionally, it has been observed that the size of the formulas and in turn also the computation time grows depending on the number of constants defined by the enumeration type. The computation runtime actually grows approximately exponentially in the number of constants. Hence, the conformance check with enabled equivalence checks may be used if it is possible to abstract away details and use sets of enumeration constants with only a low number of available constants. This could for instance be achieved by grouping sets of values into equivalence classes and representing each class through one constant. A similar approach has been followed by Aichernig et al. to cope with the state space explosion

# constants	creation of execution tree
2	$8.61s$
3	$21.44s$
4	$285.67s$
5	$3228.80s$
6	$75660.97s$

Table 7.9: Runtime for the creation of the pruned symbolic execution tree of the set-buffer up to depth five with varying number of enumeration constants.

problem [4]. Since it is a measure against state space explosion, it actually mitigates the benefits of symbolic execution. Nevertheless, the computation runtimes for the creation of symbolic execution trees up to depth five with varying number of enumeration constants are given in Table 7.9.

The problems related to runtime have now been discussed. Despite facing efficiency problems, the conformance check works conceptionally. In other words, it is possible to generate test cases from models involving complex data types. It should be noted that the detection of quiescence does not cause problems, although, like checking state equivalence, it involves negated existential quantification of action parameters.

Summary. This chapter discussed several case studies. Two of them are based on use cases provided by industrial partners AVL and Ford within previous projects. The **sioco** conformance checker performed well on those examples and thus is able to handle real-world models. Furthermore, the case studies showed that the symbolic approach is able to efficiently generate test cases as compared to a concrete approach, especially when actions with large parameter spaces are involved.

Additionally, the experiments with models translated from timed automata showed that the symbolic execution approach may be well-suited for checking conformance between models of real-time systems. However, while the SMT-solver Z3 is able to efficiently check satisfiability of formulas using linear arithmetic and quantifiers, as required for real-time models, it may not be able to decide whether a formula involving arrays and quantification is satisfiable. Hence, it faces efficiency problems when checking conformance of models using complex data types. Nevertheless, the conformance checker is still able to handle such models, but only up to a low search depth.

8 Extensions and Adaptations

Since the experiments presented in Chapter 7 showed that the symbolic execution approach to conformance checking is efficient, investigating its applicability for further areas was found to be worthwhile. Adaptations and extensions needed to be developed for this purpose. Additionally, the existing test case generator was adapted and extended in order to improve its support of the action system modelling formalism. Concrete measures for instance allow for more convenient modelling.

This section will start with a discussion of changes of the **sioco** checker. Afterwards, two application areas other than conformance checking of action systems will be investigated. More concretely, the conformance checking of models of real-time systems and of object-oriented action systems will be discussed. The goal of the extension to other types of models is to provide efficient conformance checks for model-based mutation testing.

The support of real-time system models is motivated by the promising measurement results presented in Section 7.3.3, which compares the symbolic execution approach to a bounded model-checking approach [10]. The motivation for checking conformance between object-oriented action systems stems from previous projects, in which, among others, Bernhard Aichernig was involved. Within the projects MOGENTES[1], MBAT[2], TRUFAL[3], CRYSTAL[4] and TRUCONF[5], research leading to the development of and the actual implementation of the MoMut::UML-toolchain[6] have been performed [63]. This toolchain belongs to a family of test-case generation tools, MoMut, implementing model-based mutation testing for various types of models such as object-oriented action systems. The predecessor of the **sioco** checker, a concrete **ioco** checker, is actually a part of the Momut::UML-toolchain. Supporting conformance checking of object-oriented action systems would allow the utilisation of the toolchain and thereby allow for test case generation from UML-models, because they can be translated into object-oriented action systems.

The presented changes and extensions have either partially or fully been implemented at the time of writing, but were not the main focus of this thesis. Consequently, their current implementation status will be described.

8.1 Changes of the sioco Conformance Checker

8.1.1 Multiple Actions with Same Label

In the context of the car alarm system and particle counter experiments, it has been pointed out that the lack of nested guarded commands requires the introduction of unique identifiers, which are appended to the action labels. This changes the interface of the system on model-level, as it creates several distinguishable actions corresponding to the same event.

The second well-definedness condition given for action systems in Section 2.3.2 shall be lifted to mitigate this problem. Hence, it should be possible to define several actions with the same label. As a result, the following changes need to be implemented:

1. The $exec$-function needs to be changed in order to execute all actions with some common label. Let $s_1, s_2, \ldots, s_n$ be the results of the original $exec$-function applied for all actions corresponding to some label. The redefinition of the $exec$-function needs to form a union over all s_i, thus each action needs to be applied for all symbolic states currently considered.

[1] `http://www.mogentes.eu` (last visit: 19.11.2015)
[2] `http://www.mbat-artemis.eu` (last visit: 19.11.2015)
[3] `https://trufal.wordpress.com` (last visit: 19.11.2015)
[4] `http://www.crystal-artemis.eu` (last visit: 19.11.2015)
[5] `http://truconf.ist.tugraz.at/` (last visit: 19.11.2015)
[6] `http://www.momut.org` (last visit: 19.11.2015)

2. The $exec_{neg}$-function needs to be changed: it has to create a disjunction over the guards of all actions with some common input action label, negate it and use this condition in conjunction with the condition formed from internal action guards.
3. The non-conformance condition given by Definition 3.14 needs to be adapted as well: in addition to the disjunction over all symbolic states, a disjunction over all guards corresponding to some output action label needs to be formed.

Additionally, some of the proofs given in Chapter 4 need to be redone. The change also influences the optimisation, which checks if a mutation weakens the guard of an input action (see Section 4.10), as this optimisation is only applicable for deterministic action systems. After lifting the second well-definedness condition, non-determinism may be expressed without internal actions, thus the check determining whether a model is deterministic needs to take this into account. Furthermore, the complexity of syntactic mutation analysis rises.

By the time of writing this thesis, this change has already been implemented and first experiments have been performed. The experiments were performed with adapted versions of the particle counter and the car alarm system model with indexed labels. More concretely, the adaptations removed the indexes appended to the labels. Measurement results obtained within the experiments show that the performance of the conformance checker stays approximately the same. Hence, the ability to define several actions with the same label allows for more convenient modelling without incurring a significant performance loss.

8.1.2 Parameters for Internal Actions

The second change of the **sioco** conformance checker lifts the first well-definedness condition of action systems and thereby allows for the definition of parameters for internal actions. It is actually motivated by the planned integration into the MoMut::UML-toolchain. This toolchain creates internal actions with parameters during the translation of UML-models into object-oriented action systems. Since conformance checking of object-oriented action systems is planned to be implemented by translating object-oriented into simple action systems, the latter are required to support internal action parameters as well.

From a modelling point of view, internal action parameters allow for more concise modelling of non-observable state changes. If several internal actions share a similar form, they can be combined by introducing parameters, which are non-deterministically chosen and determine the actual state change.

Conversely, an internal action with instantiated parameters represents an internal action without parameters. Hence, from a conformance checking point of view, it would be possible to retrieve all possible parameter instantiations and thereby implicitly create sets of internal actions for one internal action. However, in the symbolic approach, it is the goal to avoid enumeration of parameters. In order to understand how to avoid enumeration, the effects of it shall be further investigated.

The set of symbolic states reachable by executing a trace of observable actions grows with the number of internal action parameter instantiations. As the Φ_{AS}-function used in the (non-)conformance condition forms a disjunction over all symbolic states, this disjunction grows as well. Since disjunction and existential quantification are related, the size of the disjunction can be reduced by avoiding the enumeration of internal action parameters. Instead, the symbolic internal action parameters should rather be existentially quantified in the Φ_{AS}-function. This existential quantification actually reflects the intuition that an internal action should be executed for some non-deterministically chosen set of parameters.

From an implementation point of view, a second type of variable index needs to be introduced. The reason is that one internal action may be executed several times in a row and their parameters need to be distinguished. Consequently, parameters of internal actions need to be indexed twice, whereby the first index corresponds to the length of the observable trace, which has been executed before. Secondly, another index needs to be added to the parameters, which corresponds to the number of internal actions, that have been executed since the last observable. Furthermore, the Φ_{AS}-function needs to be extended

by an existential quantification as noted above. Considering the non-conformance condition for product states given by Definition 3.14, the internal action parameters need to be existentially quantified for each symbolic state in a compound state separately.

Although the possibility of utilising internal action parameters allows for more convenient and concise modelling of non-deterministic state changes, it should be used with care. Internal action parameters introduce negated existential quantifiers into the non-conformance condition and thereby increase the complexity of checking satisfiability of this condition. As a result, the performance of the conformance check may be reduced. If used excessively, the symbolic approach may even be infeasible, if the SMT-solver is not able to decide satisfiability.

This extension of the conformance check has already been implemented and first experiments showed that it works correctly in principle. The **sioco** checker is thus able check conformance of models involving internal action parameters and does not produce spurious counterexamples to conformance.

8.2 Conformance Checking of Real-Time System Models

The following discussion is based on joint work undertaken together with Bernhard Aichernig and Florian Lorber. Results of the work have been submitted for publication in the Festschrift in honor of Frank S. de Boer. At the time of writing this thesis, the paper has been accepted but not yet published [11]. The discussion is based on this paper, but has a slightly different focus.

Motivated by the measurement results given in Section 7.3.3 for real-time models, the action system formalism was adapted to account for time. This led to the definition of *timed action systems*. Additionally, the **sioco** checker has been adapted as well in order to be able to process the new type of models. The Symbolic Timed Input Output Conformance (**stioco**) conformance relation defined by von Styp et al. [81] served as a starting point to check conformance of timed action systems. It is a symbolic version of the **tioco** conformance relation and inspired by **sioco**. An implementation model conforms to a specification with respect to **tioco** if it only produces observations allowed by the specification after all traces of the specification. The observation of quiescence is replaced by the observation of the passage of time in this context. This observation is usually referred to as *elapse* [64].

Hence, it is similar to **sioco** and the **sioco** conformance checker can be adapted with little effort to check **stioco** conformance. Furthermore, **tioco** is used as conformance relation by the bounded model-checking approach [10] which served as a benchmark for comparison in Section 7.3.3. This allows a more appropriate comparison of runtimes.

In correspondence to timed automata, the timed version of action systems was extended by:

1. a set of real- or integer-valued clock variables,
2. a time invariant, which defines constraints for the passage of time depending on the system's discrete state,
3. time guards, which are conditions associated with actions formed over clocks and state variables, and
4. clock reset sets. These are sets of clocks that should be reset after executing some action.

Apart from that, timed action systems consist of the same parts as simple action systems. As a result, it is possible to define data variables. As noted above, state variables may be referenced in the time invariant and also in time guards, thus timed action systems are more flexible than traditional timed automata, which do not allow for data variables. Von Styp et al. lift this restriction by introducing symbolic timed automata [81]. There also exist other approaches to circumvent this restriction, UPPAAL [65] for instance supports timed automata extended with data variables. Since timed action systems allow the definition of several actions with the same label, it is possible to translate timed automata into action systems, which behave exactly the same and provide the exact same interface.

Furthermore, symbolic execution semantics have been defined for timed action systems by the author of this thesis. These semantics require that delays and discrete actions interleave. This resembles the notion of timed traces used by the bounded model-checking approach [10] and the symbolic trace semantics defined by von Styp et al. [81]. Based on these semantics, an adapted variant of the **stioco** conformance relation and all required concepts and predicates have been defined for timed action systems.

Beside applying to timed action systems rather than to symbolic timed automata, the developed semantics and conformance relation allow for the utilisation of internal actions, which are not supported by symbolic timed automata. Therefore, the required adaptations shall be discussed shortly. In order to transform timed traces into observable traces, consecutive delays only separated by internal actions are summed up and the internal actions are removed from the trace [64]. Consequently, constraints specifying that observable delays must be equal to the sum of such consecutive *unobservable* delays need to be added to the path condition for symbolic execution. Furthermore, the *elapse* of time needs to be handled appropriately by the actual check for non-conformance. Time may elapse for a few time units, then an internal action may be executed and afterwards time may elapse again. Hence, a symbolic variant of the *elapse*-function (see Krichen and Tripakis [64]) needs to compute a τ-closure with interleaved delays essentially.

Given these considerations, the conformance checker was adapted. The **stioco** checker for instance enforces the alternating execution of actions and delays. As a result, angelic completion can be performed in a way, such that the problems described in Section 7.3.3 do not occur. Furthermore, the mutation component was changed as well and mutation operators similar to those used for timed automata [10] have been implemented.

After implementing the changes, the timed automata models discussed before have been translated again, but into timed action systems and the measurements have been performed anew with these models. Additionally, a deterministic timed automata model of a car alarm system with a PIN code has been translated as well. The PIN code needs to be sent when the car is locked or unlocked. This model serves to evaluate the handling of data variables. In contrast to the models considered in Section 7.3.3, which define five real-valued clock variables each, the car alarm system with PIN uses only one clock.

Comparisons between the bounded model-checking and the symbolic execution approach revealed that symbolic execution does indeed pay off. However, symbolic execution was still slower on average for the deterministic model with a mean conformance checking runtime of 1.65 seconds. Similar to the measurement results given in Section 7.3.3, the median of the conformance check runtimes has been very low when symbolic execution was performed.

Furthermore, the **stioco** checker was again able to process the complete non-deterministic model, but two mutants had to be excluded, as the maximum amount of available RAM was exceeded during the measurements. However, manual inspection showed that these mutants conform to the specification. For the rest of the mutants, the conformance check took on average 0.63 seconds. It should be noted that processing a single mutant took up to 230.47 seconds, thus outliers can significantly influence the average runtime needed to check non-deterministic models.

The measurements based on the car alarm system model with PIN code showed data variables do not influence the performance of either approach. However, it revealed that the runtime of the symbolic execution approach heavily depends on the number of clocks, as conformance checks took only 0.17 seconds on average, while the deterministic model without PIN needed 1.65 seconds averagely. This can be attributed to the fact that the model with PIN code defines only one clock in contrast to the five clocks defined by the other deterministic model. The runtime of bounded model-checking on the other hand was only slightly reduced by the reduction of clocks.

All conformance checks have been carried out with search depth 12, that is, search was stopped after executing 12 discrete actions.

It can be concluded that symbolic execution is an efficient approach to conformance checking of real-time system models. This becomes apparent especially when non-determinism is involved and only a low number of clocks is required.

8.3 Integration into MoMut::UML-Toolchain

As noted above, the predecessor of the **sioco** checker, a concrete **ioco** checker, is actually a part of the MoMut::UML-Toolchain[7], more precisely of the symbolic back-end provided by the toolchain. The toolchain transforms models and their mutants between various types of formalisms and then produces test cases via a mutation-based generation strategy.

More specifically, a UML-model and their corresponding mutants may be transformed into object-oriented action systems and then into non-object-oriented action systems [63]. From the model and mutants given as action systems, test cases can be generated via the symbolic back-end[8].

An integration of the **sioco** checker into this toolchain as an alternative back-end would allow to use existing experiments and to compare the different back-ends more accurately. Furthermore, it would enable users to create UML-models and to generate symbolic test cases from these models.

Since there are various transformation steps, there exist several possibilities for integrating the **sioco** check. The most promising approach is to transform object-oriented action systems into simple action systems. As there also exist more complex non-object-oriented action systems, models which can be processed by the **sioco** checker will be referred to as simple action systems for the remainder of this subsection. The other formalism will be referred to as complex action systems.

The mentioned approach allows to make use of an existing compiler Argos, which is able to translate object-oriented into complex action systems. Argos has been implemented by Willibald Krenn within the MOGENTES project. Consequently, the most natural approach would be to implement a new back-end for the compiler, which generates simple action systems. However, it is not necessary to support object-oriented action systems in the form defined by Bonsangue et al. [23]. The MoMut::UML-toolchain supports only a limited form described by Tiran in the Argos manual [75]. Limitations include the restriction to non-recursive methods and the requirement that objects may only be created at system start.

Implementation

In order to implement the translation to simple action systems correctly, the concepts provided by object-oriented action systems need to be supported. These can be grouped into two categories:

1. Object-orientation
2. Syntactical elements for composing statements

Before discussing object-orientation, types of statements shall be discussed. Like simple action systems, object-oriented action systems support assignments. They further allow for three types of composition: sequential, non-deterministic and prioritised. Additionally, a guard may be associated with a statement. This applies to bodies of methods and actions. The latter may be combined via composition operators as well in the so-called *do-od*-block. This has the effect that actions are not chosen non-deterministically during execution but the choice also respects the constraints defined by the composition. Hence, it is for instance possible to state that some action must always be executed after some other action.

These types of statements are also provided by complex action systems. In order to be able to perform refinement checks between specifications and mutants, they are normalised prior to the conformance check [9]. This normalisation step creates action systems similar in structure to simple action systems, but with sequential composition of assignments. Sequential composition, however, is also eliminated through symbolic execution.

[7] `http://www.momut.org` (last visit: 19.11.2015)

[8] The concrete **ioco** checker uses an SMT-solver, but only to retrieve parameter instantiations satisfying guards.

Hence, a similar approach could be used to translate object-oriented into simple action systems. It has already partly been implemented by means of symbolic execution. More concretely, all paths through the *do-od*-block should be symbolically executed and thereby constraints and associated symbolic states collected. Based on these, it is possible to create simple action systems, which behave exactly the same as the original action systems. It should be noted that functions, procedures and methods have not been discussed, but they could be inlined, as described by Jöbstl in her dissertation [58].

She further added limited support for object-orientation in complex action systems, such as support for method calls. But as simple action systems should be integrated via translation without changing the syntax or semantics of simple action systems, a different approach needs to be followed. Classes and objects need to be translated using provided data types.

This can be achieved based on the following steps:

1. A record definition should be created for each class. Such a record should define one field per attribute of the class.
2. An enumeration type should be created for each class. It should define one object-identifier per created object and a special *null*-constant. Note that this is possible, because objects are created statically at system start. A similar concept has been implemented for complex action systems [58].
3. Define one map state variable per class, which maps object-identifiers to records holding the actual values of the objects.
4. Each reference to an object should be replaced by its corresponding object-identifier and references to *null* should be replaced by the *null*-constant of the corresponding enumeration type.
5. Each reading from an attribute should be translated into a map lookup for the record corresponding to the object and a read from the respective record-field.
6. Each writing to an attribute should be translated into a map-update of the record corresponding to the object. The update should insert a new record into the map, which is equivalent to the old record, but the referenced attribute should be set accordingly.

These steps have also partly been implemented, but the implementation has been discontinued. As can be inferred from the experiments involving complex data types discussed in Section 7.4, the current implementation of the **sioco** checker would not be able to efficiently check action systems created this way.

Another fact which aggravates the difficulty of checking simple action systems translated from object-oriented action systems is that first-order mutants may be turned into higher-order mutants by the translation.

In conclusion, conformance checks of automatically translated object-oriented action systems are not feasible at the moment due to the problems associated with complex data types.

9 Conclusion

9.1 Summary

This thesis presented a symbolic execution-based approach to Input Output Conformance checking of action system models. The **sioco** conformance relation and related concepts defined by Frantzen et al. for STSs [45] have been adapted to action systems for this purpose. Additionally, further concepts relevant to **sioco** conformance checking have been defined. The relation between the concepts developed by Frantzen et al. [45] and those developed in the course of this thesis has been discussed formally. Hence, a theoretical foundation for checking of **sioco** conformance has been given.

A technology-independent implementation of an **sioco** checker has been described afterwards. This **sioco** checker essentially explores a product graph formed through simultaneous symbolic execution of action systems. Several optimisations building upon on the basic implementation have been defined. If possible, proofs have been given to show that these optimisations do not alter the conformance checking result.

Based on the given technology-independent description, a Scala-implementation of the conformance checker has been developed in the course of this thesis. This implementation uses the SMT-solver Z3 developed by Microsoft for satisfiability checking [37]. It is integrated in a mutation-based test case generator which has also been implemented during this thesis. The complete implementation comprises the conformance checking component, a mutation component and a test driver, which has been discussed in a technology-independent way as well.

Furthermore, two application areas of the conformance checker, mutation-based test case generation and conformance verification, have been discussed in general. However, the main focus was laid on test case generation and therefore, the testing process has been discussed more thoroughly.

Since one of the goals was to develop an efficient conformance checker, evidence has been given by means of case studies that indicate that the developed conformance checker is indeed efficient in terms of runtime. Two of the presented case studies are industrial use cases provided by AVL and Ford respectively, which have served as benchmarks for test case generation before. As a result, it was possible to compare the **sioco** checker to a concrete **ioco** checker, which has also been implemented by the author of this thesis. In addition to the comparative case studies, case studies focusing on other aspects, such as stepwise development of test models and difficulties associated with complex data types, have been discussed as well.

Since the approach was found to be efficient in many cases, several extensions and adaptations have been investigated and implemented. These extensions include the support for more convenient modelling and for conformance checking of real-time system. Action systems involving object-orientation have also been examined.

9.2 Related Work

While some related work was already discussed throughout the previous chapters, this section discusses work in the fields touched by this thesis more thoroughly. The three main influences for this thesis were the symbolic framework for model-based testing developed by Frantzen et al. [45], **ioco** conformance checking based on product graphs, which was introduced by Weiglhofer and Wotawa [83], and work in the area of model-based mutation testing.

The symbolic framework introduced concepts, which were adapted for the thesis. Furthermore, the framework laid the foundation for theoretical concepts developed within the thesis and thereby also heavily influenced the style of presentation of these concepts. Most important of all, the **sioco** conformance relation forms the basis of the conformance check.

To the knowledge of the author of this thesis, the conformance checker presented within this thesis is the first implementation of a fully symbolic **ioco** checker. The **sioco** conformance relation, however, has already been used for model-based testing by Frantzen et al. [43]. But they follow a different approach, as they perform testing randomly and on-the-fly. The theoretical foundation for the applied on-the-fly testing algorithm was also given by Frantzen et al. [44]. While the approach presented in previous chapters also foresees an on-the-fly testing phase, the testing is driven depending on a symbolic test case. Furthermore, the thesis focused on the generation of such test cases.

The symbolic framework inspired the work of Bentakouk et al. [17] as well. They also perform on-the-fly testing based on STSs, but use a different conformance relation and target service orchestration testing. Like Frantzen et al., they perform tests on-the-fly in order to be able to react to outputs from the SUT as they are observed. This way it is possible to appropriately carry on with test case execution. An alternative approach would need to enumerate all possible outputs beforehand and thus suffer from state space explosion. Similar considerations led to the choice of an on-the-fly execution strategy for the implementation of the test driver presented in Section 5.1.2.

The state inclusion criterion defined by Gaston et al. had a significant impact on the conformance checker as well [47]. It has been used to derive conditions for symbolic state equivalence and thereby built the basis for pruning of the search tree, which is explored during test case generation. Actually, Gaston et al. also employed an adapted variant of the **ioco** conformance relation together with symbolic specifications. They select finite behaviours to be tested based on test purposes. Their notion of test purposes corresponds to the test case selection strategy used in this thesis. In contrast to their approach however, this thesis suggests to base the selection of concrete inputs for testing on both selected behaviours and additional conditions. In subsequent work, the methodology has been extended to handle component-based system specifications appropriately [42]. More recently, the IOSTS-framework has been extended by Boudhiba et al. to allow for user-defined program calls in expressions [25].

The notion of test purposes has originally been introduced by Jard and Jéron [55] in the context of **ioco**-based non-symbolic test case generation through the TGV tool. Unlike the test purposes defined by Gaston et al. [47], which are given in an abstract manner independent of the actual system, test purposes accepted by the TGV tool are defined in dependence of the system. More concretely, they are given through deterministic and complete IOLTSs together with two sets containing *Accept* and *Refuse* states.

Since test purposes can be defined with respect to the modelled system, mutants can be seen as test purposes. This has been shown by Aichernig and Corrales Delgado [5]. The strategy applied in this thesis can be compared to the second killing strategy given by Aichernig et al. [4], with the difference that only at most one test case per mutant is generated. It should be noted that they describe a weakness of the strategy. It is not well-suited for systems showing highly non-deterministic behaviour. The problem is that many test case executions would issue *inconclusive*-verdicts for such systems.

The approaches to conformance checking and test case generation described in [3, 4, 27] can be seen the most important influences on this thesis, except that this thesis does not deal with continuous behaviour. Similarly to the **sioco** checker, they create product graphs from traces of action systems and search for *unsafe* states, which denote non-conforming behaviour.

The translation of action systems was inspired by SMT-solver-based model-based mutation testing. Aichernig et al. gave a predicative semantics for action systems in order to generate tests from mutated models via refinement checking [9]. Beside a translation into constraint satisfaction problems, another translation from action systems into SMT-formulas has been developed for these semantics [7]. An SMT-based translation has also been implemented for Jöbstl's dissertation [58], which served as a basis for the concrete **ioco** checker preceding the **sioco** checker presented in this thesis.

The experiments discussed in Section 7.3.4 showed that conformance verification by means of symbolic execution is feasible. A symbolic execution-based approach to conformance testing has also been examined by Le Gall et al. [46]. However, they combine symbolic with concrete execution and check for refinement.

9.3 Discussion

9.3.1 Development

As indicated above, to the knowledge of the author of this thesis, the first fully symbolic **ioco** checker has been developed formally and implemented in the course of this thesis. This can be seen as the main contributions of this thesis. The aspects involved in the development shall be discussed briefly in the following.

Formal development. The formal representation of symbolic execution and conformance checking concepts via first-order logic turned out to be advantageous. It allowed to formally reason about the utilised techniques. Thereby, it was possible to show that the followed approach indeed checks for **sioco** conformance. This has been done by relating the basis of the conformance checking algorithm to the definition of the **sioco** conformance relation given by Frantzen et al. [45].

Optimisations. During the development, it became apparent that optimisation is crucial for symbolic exeution to be efficient. This is reflected in the large number of optimisations presented in this thesis. These optimisations focus either on symbolic execution or mutation-based test case generation. They do not include generally applicable techniques like caching, although the actual implementation makes use of such techniques. The formal description of optimisations permitted to prove that their utilisation does not introduce unwanted behaviour.

Implementation. The formally described **sioco** conformance checker has been implemented in Scala during this thesis. It uses the SMT-solver Z3 [37] for satisfiability checking and serves to demonstrate that mutation-based test case generation via symbolic execution and **ioco** checking is feasible.

9.3.2 Evaluation

Another contribution of this thesis is the evaluation of the proposed conformance checking approach. The evaluation has been carried out by performing of several case studies highlighting different aspects of conformance checking. Most importantly, a comparison between the implemented symbolic **ioco** checker and a concrete **ioco** checker showed that the symbolic approach is significantly faster than the concrete approach. **Symbolic test case generation is for instance on average nearly 70 times faster than concrete test case generation for the particle counter use case provided by AVL.**

Furthermore, the case studies involved models of real-time systems and also experiments concerning stepwise development of test models. Both areas were found to be amenable to a symbolic execution-based conformance checking approach.

However, a problem of the symbolic approach has been identified as well. Currently, it is not possible to perform conformance checks with large search depths for models with complex data types. This is due to the fact that the technique employed to overcome the path explosion problem involves checking satisfiability of quantified first-order formulas. This actually emphasises the importance of optimisation since disabling two of the optimisations almost rendered the approach infeasible.

9.3.3 Concluding Remarks

In conclusion, the two main goals set for this thesis have been achieved:

1. a test case generator which overcomes the state space explosion problem by symbolic handling of data has successfully been implemented.

2. the symbolic conformance checker has been compared to a concrete conformance checker with respect to runtime, which revealed that the symbolic approach pays off.

As a result of the successful achievement of these goals, the most important developments and findings of this thesis have been presented at the USE-workshop [12].

Additionally, the discussed approach has been extended, most notably to the area of conformance checking of real-time system models. A paper covering, but not limited to, this extension has been written in joint work with Bernhard Aichernig and Florian Lorber [11]. The paper has been accepted for publication but has not been published at the time of writing this thesis.

9.4 Future Work

In the following, some further extensions and adaptations of the presented approach will be discussed. They could for instance broaden the applicability of the **sioco** checker to more complex models or enable the detection of additional types of errors.

9.4.1 Heuristic Methods to Tackle the Path Explosion Problem

Cadar and Sen group methods to overcome the path explosion problem into two categories [32], heuristic methods and those based on sound program analysis. They further note that the latter group of techniques passes complexity from the exploration to the constraint solver and can therefore constitute a performance bottleneck. This observation has actually been confirmed during an analysis performed with VisualVM[1], which showed that symbolic state equivalence checks contribute a large portion of the overall runtime. Even worse, as discussed in Section 7.4, the technique of pruning redundant paths has proved to be infeasible for complex data types.

Consequently, other methods to cope with this problem should be investigated. Cadar and Sen for instance state that paths may be chosen/discarded at random, or may be chosen such that certain coverage criteria are maximised [32].

Experiments presented in Chapter 7 showed that precomputation is a vital part of the conformance check and the experiments discussed in Section 7.4 further showed that the path explosion problem already affects this part. Hence, a heuristic method could be used to prune the symbolic execution tree and thereby reduce the number of traces explored during the conformance check. Stated differently, heuristic methods may be used to select *good* traces such that it is not necessary to execute all of them.

Techniques from the area of symbolic program execution could be adapted for this purpose. Burnim and Sen [28][2] and Cadar et al. [30] for instance propose the adoption of random searches. The random selection of branches could be implemented as a random selection of actions. More concretely, at each step during the creation of the symbolic execution tree, one of the enabled actions could be chosen at random. In order to create trees rather than mere traces, the execution could be restarted at random.

Other heuristics are adaptable as well: Cadar et al. for instance also suggest that searches along paths, which recently covered new code, should be favoured and thus be continued [30]. As the structure of action systems is different from programs, it may be necessary to use another coverage criterion in combination. It may be necessary to cover one action several times in different states, thus a state-based criterion may be suitable.

Conversely, Gaston et al. [47] give a criterion based on symbolic state inclusion, which defines when symbolic execution may be stopped. However, this criterion is not well-suited as it requires checking of state inclusion, which is essentially as complex as checking of state equivalence and this operation has been identified to be infeasible for models with complex data types.

[1] https://visualvm.java.net/ (last visit: 4.10.2015)

[2] It should be noted that Burnim and Sen actually use Concolic execution, a combination of concrete and symbolic execution. However, the search strategies are applicable nonetheless.

Subpath-guided path exploration, as defined by Li et al. [67], may also be well-suited. A complete path represents a complete symbolic execution from the initial state to some end state, that is, it is a sequence of branch conditions visited along the execution. In this context, a length-n subpath is a subsequence of a complete path of length n. Paths could be represented by a sequence of action labels for the symbolic execution of action systems.

The basic idea of subpath-guided path exploration is to favour subpaths of some fixed length, which have been explored the lowest number of times. At each step, a set of possible subpaths could be computed from the set of enabled actions. The action corresponding to the subpath, which has been travelled the least, could be executed next.

However, none of these heuristics have been implemented so far. But the support of such techniques seems to be essential for the successful test case generation from models involving complex data types.

Finally, it should be noted that care has to be taken when non-determinism is involved. Since the symbolic execution tree is used during the conformance check to compute all states reachable by some trace of observable actions, internal actions need to be taken into account. This means that if an observable action a is performed for some state, it must be performed for all states reachable by the same observable trace σ. Otherwise it would not be possible to determine the set of states reachable by executing $\sigma \cdot a$. It follows that internal actions need to be executed and must not be discarded at all.

9.4.2 Livelock Quiescence

The **sioco** checking algorithm may be extended such that another class of non-conformance conditions can be detected. Tretmans defines **ioco** for IOLTSs not containing livelocks [78], which are loops of internal transitions. A similar restriction is placed on the STSs considered by Frantzen et al. [44], which states that STSs must not contain loops of internal actions as well. This restriction is stricter than the restriction given for LTSs because the LTS-interpretation of STSs containing such loops does not necessarily contain livelocks.

A possible requirement for action systems would be to demand that the IOLTS-interpretation of action systems must not contain livelocks. However, it is problematic to place such a restriction on action systems considered in this thesis. While it is a reasonable restriction for specifications, mutants cannot be guaranteed to adhere to this restriction, because mutants are generated automatically. Hence, the effects of livelocks on the conformance check should be considered.

There already exist approaches to handle livelocks, which involve quiescence. States connected by a loop of internal actions are treated as quiescent states by Jard and Jéron [55], thus self-loops labelled δ are added to those states. The intuition behind this approach is that systems with livelocks may perform internal actions for an infinite amount of time without producing any outputs. Hence, such systems would appear as quiescent.

The notion of Divergent Quiescent Transition Systems(DQTSs) has been introduced by Stokkink et al. as an alternative to Suspension Automata [73], which are for instance used by TGV [55]. With DQTSs it is possible to model quiescence more appropriately in the presence of divergence, that is, if there exist paths of infinite length containing only internal actions. The treatment of livelocks proposed in the following is, however, based on the approach followed by TGV, as it is suited to be used in conjunction with action systems.

Example 3.2 shows that the interpretation of a symbolic state may contain concrete states involved in a livelock on IOLTS-level and it may also contain concrete states which are not involved in a livelock. Hence, quiescence resulting from livelocks depends not only on the state but also on an additional condition. An algorithm shall be sketched in the following which detects such quiescence conditions.

As a prerequisite, the τ-closure needs to compute all traces of internal actions and keep track of the positions of symbolic states within these traces. After computing the τ-closure, all pairs of symbolic states $\eta_1 = (\varphi_1, \rho_1)$ and $\eta_2 = (\varphi_2, \rho_2)$, which lie on the same trace such that η_1 is reached before η_2, shall be examined. Given such a pair (η_1, η_2), the following condition shall be formed: $\chi = \varphi_1 \wedge \bigwedge_{x \in \mathcal{V}} x =$

$\rho_1(x) \wedge \varphi_2 \wedge \bigwedge_{x \in \mathcal{V}} x = \rho_2(x)$. The condition expresses that both symbolic states must be reachable and their symbolic state vectors must be equal, that is, it must be possible to reach the same state again by executing internal actions.

There exists a livelock if χ is satisfiable. More concretely, the livelocks exist for all valuations $\upsilon \in \mathfrak{U}^{\mathcal{V}}$ for which there exists a valuation $\zeta \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ with $\upsilon \cup \zeta \models \chi$. Hence, χ is a condition for quiescence. The disjunction over all conditions $\overline{\exists}_{\mathcal{V}} \chi$ formed for all pairs of states gives a characterisation of all possible situations, in which quiescence may arise from livelocks in a τ-closure. Consequently, this condition should be added to the quiescence condition Δ via disjunction. The resulting condition would check quiescence arising from the absence of outputs and from livelocks simultaneously.

Since this extension has neither been tested nor implemented, such an approach to detecting livelocks may prove to be infeasible, as it requires a large number of satisfiability checks.

9.4.3 Further Extensions

Several other extensions come to mind, when considering different steps in the test case generation process. Some of them will be examined briefly in the following.

Improved Test Case Extraction and Killing Strategies

Currently, counterexamples to conformance are used as test cases and inconclusive-verdicts are added during an online-testing phase. Furthermore, only one test case is retrieved per non-conforming mutant. Aichernig et al. propose several *killing strategies* generating varying numbers of tests, different types of tests and different sets of tests per mutant [4]. The killing strategies essentially define search strategies. One strategy for instance generates test cases for all traces leading to all different unsafe states. Another killing strategy generates adaptive test cases. These are test cases which list several paths to an unsafe state if possible. As a result, such test cases only give an inconclusive-verdict after observing an output a, if it is not possible to reach the unsafe state s by executing a. The state s is the state to be covered by the respective test case.

Since these test case generation strategies require techniques such as a backward search from an unsafe state to the initial state, it may be difficult to implement them symbolically. Nevertheless, adaptations of them may improve the fault detection capabilities of symbolic test cases.

Higher-order Mutants and Additional Mutation Operators

Higher-order mutants are mutants containing several faults. Stated differently, higher-order mutants combine several first-order mutants. Jia and Harman motivate the study of higher-order mutants by the fact that there are higher-order mutants denoting faults more subtle than those denoted by first-order mutants [57]. They show that there indeed exist so-called *subsuming* higher-order mutants [56]. These are mutants that are harder to kill than their corresponding first-order mutants. Hence, it may be worthwhile to study higher-order mutations on a model-level as well, as tests generated from such mutants may find subtle faults and also corresponding simple faults.

In order to support test-case generation based on higher-order mutants, it is necessary to adapt the mutation analysis performed within the conformance check. It is for instance necessary to check whether any of the mutated actions has been executed, before applying certain optimisations.

A comparison between the set of mutation operators applied in this thesis and the mutation operators supported by other tools suggests that additional mutation operators should be implemented. The first set of mutation operators, those supported by the Mothra system [57, 61], lists some mutation operators, which may be implemented for action systems, but are not supported at the moment. These operators may lead to the generation of a more comprehensive test suite. However, limited support of mutation

operators is not a severe restriction. Jia and Harman note that the assumption that all mutation operators of Mothra need to be considered is actually outdated [57].

Further Optimisations

While several optimisations have been implemented in the course of this thesis, there is still room for improvement. One area, which may be improved, shall serve as an example: tactics used to configure Z3 have been discussed in Section 6.4, but only very simple tactics are provided at the moment. Based on linear integer arithmetic benchmarks, Moura and Passmore have shown that more complex tactics may be more effective and may also be faster than the default SMT-solver tactic [38]. They further note that for different types of problems, different heuristics may be more efficient. Hence, it may make sense to choose tactics based on an analysis of the action system model. But prior to implementing such context-dependent tactics, a large number of experiments needs to be performed to find optimal combinations of reasoning engines.

Other Symbolic Test Case Selection Criteria

Frantzen et al. not only define a symbolic version of **ioco**, but they rather give a general symbolic framework for model-based testing [45]. Although the main focus of this thesis was the implementation of an efficient **sioco** checker, the implemented concepts may also be applied in other settings. Other test case selection criteria, such as those given by Gaston et al. [47], could be implemented. Alternatively, model-based mutation testing with refinement as conformance relation could also be implemented [9]. In both cases, the same test driver as presented in Section 5.1.2 could be used.

9.4.4 Additional Case Studies

Chapter 7 discusses a wide range of aspects of conformance checking. For this reason, application areas such as stepwise development of test models are investigated rather briefly. Hence, additional case studies highlighting different aspects should be performed. Possible case studies include, but are not limited to:

- more comprehensive case studies involving complex data types. It would be interesting to determine whether it is possible to generate test cases for non-trivial models despite facing the path explosion problem.
- case studies with large numbers of parameters. Although the refactored model of the particle counter presented in Section 7.2.3 contains actions with a large parameter space, additional case studies would be beneficial. They might demonstrate limitations of the approach or reinforce the observation that it performs well for this kind of models.
- case studies concerning stepwise development of test models. In Section 4.4, it has been pointed out that angelic completion may lead to conformance violations. While this has not been an issue for the verification of conformance between refinements of the car alarm system (see Section 7.3.4), it may be an issue considering other models. Therefore, a further investigation of this topic should be performed.

There are of course many more possibilities for further analysis of the **sioco** checker. Nevertheless, it can be concluded that the presented approach is promising.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. (Cited on pages 5, 15 and 111.)

[2] Bernhard K. Aichernig. *Model-Based Mutation Testing: Theory and Application*. Habilitation, Graz University of Technology, Institute for Software Technology, 2012. (Cited on pages 3 and 73.)

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 228–249. Springer, 2009. (Cited on pages 1, 15, 18, 55, 97 and 123.)

[4] 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*, 25(8):716–748, 2015. (Cited on pages 5, 15, 18, 73, 76, 93, 94, 97, 98, 104, 107, 115, 123 and 127.)

[5] Bernhard K. Aichernig and Carlo Corrales Delgado. From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2006. (Cited on page 123.)

[6] Bernhard K. Aichernig and Elisabeth Jöbstl. Efficient refinement checking for model-based mutation testing. In Antony Tang and Henry Muccini, editors, *QSIC 2012, 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29, 2012*, pages 21–30. IEEE, 2012. (Cited on pages 15, 45 and 48.)

[7] Bernhard K. Aichernig, Elisabeth Jöbstl, and Matthias Kegele. Incremental refinement checking for test case generation. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2013. (Cited on pages 15, 83 and 123.)

[8] Bernhard K. Aichernig, Elisabeth Jöbstl, and Martin Tappler. Does this fault lead to failure? - combining refinement and input-output conformance checking in fault-oriented test-case generation. *Journal of Logical and Algebraic Methods in Programming*, In press. Festschrift in honor of José Nuno Oliveira. (Cited on page 7.)

[9] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97:383–404, 2015. (Cited on pages 3, 4, 5, 15, 97, 98, 99, 120, 123 and 128.)

[10] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013. (Cited on pages 7, 97, 99, 104, 105, 107, 108, 116, 118 and 119.)

[11] Bernhard K. Aichernig, Florian Lorber, and Martin Tappler. Conformance checking of real-time models - symbolic execution vs. bounded model checking. In *Theory and Practice of Formal*

Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, Lecture Notes in Computer Science Festschrifts. Springer, 2016. Accepted but not yet published. (Cited on pages 7, 118 and 125.)

[12] Bernhard K. Aichernig and Martin Tappler. Symbolic input-output conformance checking for model-based mutation testing. In *Usages of Symbolic Execution - 1st International Workshop in conjunction with FM 2015, USE'15, Oslo, Norway, June 23, 2015*, Electronic Notes in Theoretical Computer Science. Elsevier B. V., 2015. In press. (Cited on pages 7, 9, 15, 21, 30, 34, 48, 79, 91 and 125.)

[13] Ralph-Johan Back and Reino Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4):513–554, 1988. (Cited on page 15.)

[14] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989. (Cited on page 15.)

[15] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010. (Cited on pages 81 and 83.)

[16] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. (Cited on page 104.)

[17] Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings*, volume 5826 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2009. (Cited on page 123.)

[18] Gilles Bernot. Testing against formal specifications: A theoretical view. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 1991. (Cited on page 2.)

[19] Juan C. Bicarregui, editor. *Proof in VDM: Case Studies*. Formal Approaches to Computing and Information Technology (FACIT). Springer-Verlag London, 1998. (Cited on page 1.)

[20] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003. (Cited on pages 29, 33 and 79.)

[21] Nikolaj Bjørner. Linear quantifier elimination as an abstract decision procedure. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2010. (Cited on page 84.)

[22] Nikolaj Bjørner. Taking satisfiability to the next level with Z3 - (abstract). In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2012. (Cited on page 81.)

[23] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 68–95. Springer, 1998. (Cited on pages 15 and 120.)

[24] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2008. (Cited on page 48.)

[25] Imen Boudhiba, Christophe Gaston, Pascale Le Gall, and Virgile Prevosto. Model-based testing from input output symbolic transition systems enriched by program calls and contracts. In Khaled El-Fakih, Gerassimos D. Barlas, and Nina Yevtushenko, editors, *Testing Software and Systems - 27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015, Proceedings*, volume 9447 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2015. (Cited on page 123.)

[26] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, April 1975. (Cited on pages 19, 21 and 22.)

[27] Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. Automated conformance verification of hybrid systems. In Ji Wang, W. K. Chan, and Fei-Ching Kuo, editors, *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pages 3–12. IEEE, 2010. (Cited on page 123.)

[28] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 443–446. IEEE, 2008. (Cited on page 125.)

[29] Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996. (Cited on page 17.)

[30] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. (Cited on page 125.)

[31] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10:1–10:38, December 2008. (Cited on page 19.)

[32] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013. (Cited on pages 19, 48, 96, 113 and 125.)

[33] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008. (Cited on page 5.)

[34] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial*

Lectures, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. (Cited on page 6.)

[35] Adenilso da Silva Simão and Alexandre Petrenko. Generating asynchronous test cases from test purposes. *Information and Software Technology*, 53(11):1252–1262, 2011. (Cited on pages 79 and 107.)

[36] Leonardo de Moura and Nikolaj Bjørner. Z3–a tutorial, 2006. (Cited on pages 82 and 84.)

[37] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. (Cited on pages 81, 122 and 124.)

[38] Leonardo Mendonça de Moura and Grant Olney Passmore. The strategy challenge in SMT solving. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2013. (Cited on pages 85 and 128.)

[39] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94, Part 2(0):130 – 143, 2014. Abstract State Machines, Alloy, B, VDM, and Z Selected and extended papers from ABZ 2012. (Cited on page 5.)

[40] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978. (Cited on page 3.)

[41] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. (Cited on page 73.)

[42] Alain Faivre, Christophe Gaston, and Pascale Le Gall. Symbolic model based testing for component oriented systems. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings*, volume 4581 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 2007. (Cited on page 123.)

[43] Lars Frantzen, Maria de las Nieves Huerta, Zsolt Gere Kiss, and Thomas Wallet. On-the-fly model-based testing of web services with Jambition. In Roberto Bruni and Karsten Wolf, editors, *Web Services and Formal Methods, 5th International Workshop, WS-FM 2008, Milan, Italy, September 4-5, 2008, Revised Selected Papers*, volume 5387 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 2008. (Cited on page 123.)

[44] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004. (Cited on pages 123 and 126.)

[45] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A symbolic framework for model-based testing. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006. (Cited on pages 6, 9, 11, 12, 13, 17, 22, 23, 24, 26, 32, 38, 52, 88, 89, 122, 124 and 128.)

[46] Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for refinement testing. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, volume 4454 of *Lecture Notes in Computer Science*, pages 131–148. Springer, 2007. (Cited on page 123.)

[47] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006. (Cited on pages 22, 27, 28, 51, 84, 123, 125 and 128.)

[48] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009. (Cited on page 84.)

[49] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioners Series. Prentice Hall, 1995. (Cited on page 5.)

[50] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 — A modern refinement checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2014. (Cited on page 5.)

[51] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005. (Cited on page 19.)

[52] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer, 1985. (Cited on page 6.)

[53] William E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, C-24(5):554–560, May 1975. (Cited on pages 19 and 21.)

[54] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004. (Cited on page 9.)

[55] Claude Jard and Thierry Jéron. TGV: Theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, August 2005. (Cited on pages 123 and 126.)

[56] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*, pages 249–258. IEEE, 2008. (Cited on page 127.)

[57] 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 3, 73, 127 and 128.)

[58] Elisabeth Jöbstl. *Model-Based Mutation Testing with Constraint and SMT Solvers*. PhD thesis, Graz University of Technology, Institute for Software Technology, 2014. (Cited on pages 1, 6, 8, 18, 88, 91, 92, 121 and 123.)

[59] Clifford B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991. (Cited on page 5.)

[60] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. (Cited on pages 19, 20, 22 and 30.)

[61] K. N. King and A. Jefferson Offutt. A Fortran language system for mutation-based software testing. *Software – Practice & Experience*, 21(7):685–718, June 1991. (Cited on pages 73 and 127.)

[62] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: integrating SMT and programming. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer, 2011. (Cited on page 83.)

[63] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard K. Aichernig, Elisabeth Jöbstl, and Harald Brandl. MoMut::UML model-based mutation testing for UML. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–8. IEEE, 2015. (Cited on pages 116 and 120.)

[64] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009. (Cited on pages 98, 104, 107, 118 and 119.)

[65] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. (Cited on page 118.)

[66] Nancy G. Leveson and Clark Savage Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993. (Cited on page 1.)

[67] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 19–32. ACM, 2013. (Cited on page 126.)

[68] Florian Lorber, Amnon Rosenmann, Dejan Nickovic, and Bernhard K. Aichernig. Bounded determinization of timed automata with silent transitions. In Sriram Sankaranarayanan and Enrico Vicario, editors, *Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings*, volume 9268 of *Lecture Notes in Computer Science*, pages 288–304. Springer, 2015. (Cited on page 109.)

[69] Paulo J. Matos and João Marques-Silva. Model checking Event-B by encoding into Alloy. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*, page 346. Springer, 2008. (Cited on page 5.)

[70] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008. (Cited on page 82.)

[71] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, January 1992. (Cited on page 73.)

[72] Hans Jürgen Ohlbach. Extensions of first-order logic, Maria Manzano. *Journal of Logic, Language and Information*, 7(3):389–391, 1998. (Cited on page 9.)

[73] Willem Gerrit Johan Stokkink, Mark Timmer, and Mariëlle Stoelinga. Divergent quiescent transition systems. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 214–231. Springer, 2013. (Cited on page 126.)

[74] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. (Cited on page 19.)

[75] Stefan Tiran. *The Argos Manual*. Institute for Software Technology (IST), Graz University of Technology, 2012. (Cited on page 120.)

[76] Stefan Tiran. On the Effects of UML Modeling Styles in Model-based Mutation Testing. Master thesis, Graz, University of Technology, 2013. (Cited on page 88.)

[77] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996. (Cited on pages 2 and 3.)

[78] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996. (Cited on pages 1, 3, 5, 11, 12, 13, 14, 23, 33, 34, 64, 67 and 126.)

[79] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. (Cited on pages 23, 24, 31, 37, 41 and 112.)

[80] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, August 2012. (Cited on pages 1, 2, 5 and 74.)

[81] Sabrina von Styp, Henrik C. Bohnenkamp, and Julien Schmaltz. A conformance testing relation for symbolic timed automata. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings*, volume 6246 of *Lecture Notes in Computer Science*, pages 243–255. Springer, 2010. (Cited on pages 118 and 119.)

[82] Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. Fault-based conformance testing in practice. *International Journal of Software and Informatics*, 3(2-3):375–411, 2009. (Cited on page 13.)

[83] Martin Weiglhofer and Franz Wotawa. "On the fly" input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 286–291, Anaheim, CA, USA, 2008. ACTA Press. (Cited on pages 32, 52, 67 and 122.)

[84] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19:1–19:36, October 2009. (Cited on page 1.)