Software quality has become an increasingly important concern in software development, as a consequence of the ever-ascending trend of using software system in critical systems. Testing as a technique for establishing a certain level of software quality has received much attention in the past decades. Model-based testing is a structured approach to testing. Using model-based testing the process of generating test-cases and predicting the correct outcome of test-cases can be mechanized. By using rigorous models for system behavior, model-based testing is formalized in terms of a mathematical notion of conformance. Input-output conformance (ioco) is a widely-studied and commonly-used conformance relation.

The ioco relation has some limitation in testing concurrent systems. Such systems are composed of interacting components which often communicate asynchronously with their environment. However, the ioco testing theory was developed based on the assumption that a tester can always communicate with an implementation under test synchronously. It is also well-known that the ioco relation does not have the compositionality property.

In this thesis theoretical foundations are developed to provide solutions to make the ioco relation suited for testing concurrent systems.
Improving Input-Output Conformance Testing Theories

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 27 oktober 2014 om 16:00 uur

door

Neda Noroozi

goingen te Mashhad, Iran
Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van
de promotiecommissie is als volgt:

voorzitter: prof.dr. B. Koren
1e promotor: prof.dr. M.R. Mousavi
2e promotor: prof.dr.ir. J.F. Groote
copromotor(en): dr.ir. T.A.C. Willemse
leden: prof.dr. M. van den Brand
        prof.dr. H. Brinksma (Universiteit Twente)
        dr. ir. G.J. Tretmans (Radboud University Nijmegen)
        prof.dr. J. Peleska (Universität Bremen)
Improving Input-Output Conformance Testing Theories

Neda Noroozi
Copyright ©2014 by Neda Noroozi
All rights reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

A catalogue record is available from Eindhoven University of Technology Library

IPA Dissertation Series 2014-13

Typeset with \LaTeX (\TeX Live 2012)
Cover design by Zahra Haririan
Printed by the print service of Eindhoven University of Technology, The Netherlands

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The author was working at Eindhoven University of Technology.
# Contents

Preface iii

1 Introduction 1
   1.1 Validation and Verification 3
   1.2 Testing 4
   1.3 Model-Based Testing 5
   1.4 Formal Conformance Testing 7
   1.5 Research Questions 8
   1.6 Outline and Origin of Chapters 9

2 Preliminaries 13
   2.1 Labeled Transition Systems 13
   2.2 Test Cases 24
   2.3 Input-Output Conformance Testing for IOLTSs 27

3 Implementing Input-Output Conformance Testing: A Case Study 31
   3.1 EFT Switch Functionality 33
   3.2 Testing the EFT Switch 34
   3.3 Test Results 43
   3.4 Discussion 44
   3.5 Closing Remarks 45

4 Implementation Relations 47
   4.1 Input-Output Conformance Relations with Quiescence 48
   4.2 The Effect of Models on the Power of Testing 49
   4.3 The Effect of Observations on the Power of Testing 68
   4.4 Test Case Generation 73
   4.5 Closing Remarks 80

5 On the Complexity of Input-Output Conformance Testing 83
   5.1 ioco Checking for Nondeterministic Models 85
   5.2 A Coinductive Definition of ioco 89
   5.3 ioco Checking of Deterministic Specifications 91
   5.4 Conformance Checking for Internal-Choice Traces 98
   5.5 Closing Remarks 102
Preface

I moved to the Netherlands four years ago to pursue my doctoral studies. As I worked on the research that has resulted in this dissertation, I have benefited greatly from the support and guidance of a number of people and I would like to take this opportunity to express my gratitude to those who helped me begin, carry out, and finish this research.

First of all, I would like to express my special thanks to Mohammad Reza Mousavi, my first promoter. He introduced me to the joys of model-based testing, helped me start my PhD research in this field and has continued to advise and guide me through these four years.

His support has not been limited to my work and I have learned a great deal from him in both my research and my personal life.

I would like to thank my second promoter Jan Friso Groote, head of the MDSE group, for giving me the opportunity to work as a PhD candidate in his group. I am very grateful for all his support. I enjoyed being part of his group, and learned many things from the discussions I had with him.

I would like to express my deepest gratitude to my daily supervisor, Tim Willemse, for his help and supervision. He contributed greatly to the details of my work, presented in this thesis.

He always provided invaluable, precise and insightful comments to improve my work; his comments were crucial to all steps. I could always walk into his office to discuss any problems I had and I really appreciated it. I learned from him many valuable things about logical reasoning, problem analysis, and structurally presenting results.

I would like to thank my co-authors for their collaboration on the papers that form the basis of this thesis. My special thanks goes to Hamid Reza Asadi and Ramtin Khosravi for their friendly cooperation and inspiring discussions we had on the project of applying model-based testing on an EFT switch. The results of that experience have motivated my PhD research. I thank Tim Willemse and Mohammad Reza Mousavi for their insightful comments and inspiring discussion on the topic of our papers.

I would like to thank the committee for reviewing my thesis: Mark van den Brand, Ed Brinksma, Barry Koren, Jan Peleska, and Jan Tretmans. I am grateful for valuable comments provided by them. I would also like to extend special thanks to Jan Ttretmans for taking time to discuss his comments in person with me.

I would like to thank the members of Formal System Analysis group, which is part of the section of Model Driven Software Engineering (MDSE). Particularly, I greatly thank my office mates, in chronological order Sjoerd Cranen, Harsh Beohar, Jeroen Keiren, Ulyana Tikhonova, and Yanja Dasjuren. I am also very grateful to my paranyphs, Ulyana
Tikhonova and Yanja Dasjuren. During the last two years of my PhD research, we had regular PhD meetings in the MDSE group to discuss our ongoing research in a friendly and informal atmosphere. I enjoyed those meetings and learned new things through them. I would like to thank other PhD students in MDSE for creating such a friendly working atmosphere and bringing about joyful occasions in our group. The secretary for our group, Tineke van den Bosch, has always been very helpful and hospitable. I would like to give my special thanks to her for all her help and for the nice chats we had.

It took some time to finish this thesis after my PhD research ended, and I thank Robert Lenders and Rachid Kherrazi for allowing me to work on it during my work in Nspyre.

I would like to thank my family. My deepest and sincerest thanks goes to my parents for their everlasting support throughout my life and for their enthusiasm about my research even though it might not have been very tangible for them. My special thanks goes to Hojjat’s parents for their support and endless encouragement.

Finally, I would like to thank my dear husband, Hojjat, for all his patience, support, and encouragement. This work would not have been possible without his care and support.

I would like to express my deep gratitude towards him for coming with me to the Netherlands despite all sort of difficulties he went through. I am very grateful to you, Hojjat, for always believing in me, even at the times when I did not. Hojjat, your endless love, support and patience mean a lot to me. I am so fortunate to have you in my life.

Neda Noroozi, September 2014
Chapter 1

Introduction

Since the emergence of computers, software has penetrated virtually every area of modern societies. Today, it is very difficult to imagine our world without software. Almost every business in every sector depends on software in its entire life cycle: development, production, marketing, and support. A significant portion of the prevalence of software is due to embedded software systems. The fast shrinking of the size of transistors (Moore’s law), the increase in mobility, and the decrease in power consumption give embedded software the opportunity to expand their presence in many different areas. Examples of various industries affected by software include the telecommunications industry, the automotive industry, the healthcare industry, the aerospace industry, and the financial industry. The volume of embedded software increases at a rate of 10 to 20 percent per year depending on the domain [EJ09].

Using software creates both huge value and unprecedented risks. Software bugs, and failures are a major vulnerability of software systems. Bugs are inherent in software in view of its (growing) complexity [Tas02]. McConnell presented in [McC04] a breakdown of the average number of bugs based on software size. The number of bugs ranges from 15-50 errors per 1000 lines of code in average industrial projects to at least 3 bugs per 1000 lines of code in some of the highest quality software. Nowadays, the complexity of software is not measured in thousand (kilo) lines of code (KLOCs), but in some million lines. For instance, a program such as Microsoft’s windows XP—with 40 million lines of code—would have had at least 20,000 bugs when launched. A concise summary of the industry standards about the quality of software in terms of the number of bugs is presented in [Osa12].

Software bugs have a wide range of impact. For instance, they can cause run-time crashes, performance degradations, incorrect program behaviors, and exploitable security vulnerabilities. They have severe consequences in safety-critical systems where an error may result in the loss of lives. The following examples are failure stories of software with a considerable impact; they demonstrate the vital importance of software quality:

1. Software bugs in defense systems.
   On June 3, 1980, and again on June 6, 1980, NORAD (North American Aerospace Defense Command) falsely reported that the US was under nuclear attack. The
problem was caused by a failure in a computer device and a faulty message design. On September 26, 1983, in a similar incident, a Soviet satellite twice reported the launch of an American missile, which was detected as a false alarm later. The false alarm was caused by a rare alignment of sunlight as one of the input data of that satellite.

2. Disasters in medical equipment.
Between 1985 to 1987, the Therac-25 medical radiation therapy device was involved in at least six incidents in which patients received a massive overdose radiation as a side effect of faulty software powering the device [LT93]. Several days after treatment, patients had signs of radiation poisoning. In three cases, patients died as a direct result of the radiation overdose.

In another radiation overdose incident, in Panama city in 2000, at least 5 of 21 patients died because of a software bug in the radiation system. The software operating the system delivered different doses depending on the order in which data was entered [Won+10; Bor06].

3. Disasters in aircrafts.
On October 7, 2008, a bug in the aircraft-monitoring system of Airbus A330-303 caused a failure in the autopilot system in a flight from Singapore to Australia. Because the pilot of the plane could switch to manual mode and control the situation, this incident fortunately resulted in no loss of lives. However, 110 people were injured in this accident.

More examples of bugs detected in software with their consequences can be found in the media and scientific publications. Software bugs are regularly published in the Software Engineering Notes magazine.

There is also a considerable financial cost to organizations using software with poor quality. For instance, because of a firmware error in pacemakers between 1990-2000, about 40 percent of the total of half a million devices were returned [EJ09]. In a 2002 report from the U.S. Commerce Department's National Institute of Science and Technology (NIST) [Tas02], it was estimated that software defects cost the US economy $59.5 billion annually. The following examples are some of the software bugs with a significant financial impact.

1. Bugs in automotive systems.
   In February 2010, Toyota reported that there was a software glitch in the braking system of its 2010 model. The automaker recalled about 6.5 million vehicles because of the problem.

2. Bugs in the financial sector.
   In August 2012, a software glitch cost Knight Capital Group, an American financial services firm, $440 million. Because of that problem, the firm was on the verge of bankruptcy.

3. Bugs in aerospace industry.
   In 1999 and 2000, software bugs caused two failures in NASA’s Mars Climate Orbiter and Mars Polar Lander. The total of the two failures cost about $357 million [Won+10].
The study in [Tas02] demonstrated that even though error-free software is not attainable, by proper testing, at least one-third of all bugs can be detected and removed in earlier phases of software development. This would result in an annual benefit of $22.2 billion to the US economy. Therefore, establishing methods and infrastructures to measure and improve the quality of software is essential. In recent years, many studies have been conducted to define software quality attributes and different metrics to assess the value of each attribute. Common software quality attributes identified in ISO 9126 [ISO01] include functionality, reliability, usability, efficiency, maintainability, and portability. In this thesis, we are mainly concerned with the functionality quality attribute.

1.1 Validation and Verification

In general, there are two different types of activities to assess software quality: validation and verification. They are considered as complementary activities in evaluating the quality of software. Besides their similarities, there is a subtle distinction between them. In IEEE standard 1012-2004 [IEE04], they are defined as follows:

- Validation: the process of evaluating a system during or at the end of the development process to determine whether it satisfies specified requirements.

- Verification: the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation activities entail checking whether a product of each phase of software development fulfills and implements the user’s intended goals. The product under evaluation is, therefore, the final code of the system or a specification of the system. A specification is an abstract model of the system that captures some aspects of the requirements of the system. With respect to the final code, validation checks whether the actual goals of the system are met by the implementation. With respect to a specification, validation involves checking if the specification can lead to a system satisfying the intended purpose of the system [YP05]. Because of checking against the actual needs of the system, validation activities necessarily need domain knowledge and human judgment [YP05; AO08]. The distinction between verification and validation is consistent with the well-known description of validation and verification given by Boehm [Boe84] as "building the right product" and "building the product right", respectively.

Verification activities involve evaluating whether an implementation is consistent with its specification. A specification for instance can be an overall design of a system, while an implementation is a design of the system with more details. In this sense, checking consistency between the products of different development phases is considered to be a verification activity. In addition, verification can check for self-consistency of a specification, for instance by proving properties on specifications, for instance, using model-checking.

Testing plays an important role in evaluating the quality of software systems. Software testing is a set of activities for finding errors in an implementation with the aim to increase quality of the developed system [MS04]. Generally speaking, an error is a deviation of the behavior of the developed system from the desired and intended goal of the system.
Chapter 1. Introduction

Testing can be considered as both verification and validation, depending on the intention of testing and the artifacts used \[\text{YP05; AO08}\]. Figure 1.1 shows the relation between verification, validation and testing activities. The diagram depicted in this figure is a variant of the well-known V-model \[\text{Boe84}\].

![Validation and verification activities in the software development](image)

Figure 1.1: Validation and verification activities in the software development

1.2 Testing

Test activities consist of designing test cases, making executable test cases, executing test cases and eventually analyzing test results \[\text{AO08}\]. Test engineers often design tests by identifying test requirements. These requirements are derived from system requirements or a specification of them that is produced at different development phases. Testing therefore can happen at different phases of the software life cycle, see Figure 1.1:

- **Unit Tests**: these are designed to assess the smallest units developed in the implementation phase. They comprise the lowest level of testing.

- **Integration Tests**: these are designed to assess the subsystem design level. They check whether interfaces between different modules are consistent.

- **System Tests**: they accompany the specification level of software development. These are designed to assess whether the assembled system as a whole fulfills its specification. Specifications are assumed to meet the requirements at the previous phase, \(i.e.,\) the requirement analysis phase.

- **Acceptance Tests**: they assess the final software with respect to users' requirements. These are designed to determine if the completed product meets the actual goals of the system. Acceptance testing involves users or other individuals with strong domain knowledge. Therefore, acceptance testing is considered to be a validation activity.
1.3 Model-Based Testing

- Regression Tests: these are used after the software system release when changes are applied to software. These check if the system still has the functionality that it had before those updates.

For each phase of software development, different attributes of software quality of a systems can be tested. For instance, robustness testing assesses reliability of a system by checking how an implementation reacts to abnormal environments. Performance testing evaluates the reliability by checking whether the implementation performs as expected. Conformance testing assesses the functionality of a system by checking whether the behavior of an implementation conforms to specified functions. In this thesis, we focus on conformance testing without concentrating on any specified abstraction level in testing.

Another classification of testing strategies is based on the levels of access to the structure of an implementation: white-box testing, gray-box testing, and black-box testing. In white-box testing (also called logic-driven or glass-box testing [MS04; Bak+07]), the internal structure of an implementation is known and used in testing. Test cases are driven from an examination of the implementation’s logic, mostly without using any kind of specification. All testing methods using coverage criteria for an implementation, e.g., statement coverage and branch coverage, are white-box testing. In black-box testing (alternatively called behavioral or functional testing [Bei95]), an implementation is only accessible through its interfaces and no knowledge about its internal structure or behavior is available. This type of testing is mostly done by executing the implementation and test results are determined based on the observable behavior of the implementation. Boundary value analysis or equivalence-class partitioning are two examples of black-box testing. Gray-box testing combines aspects of white-box testing and black-box testing. In gray-box testing, the internal structure of an implementation is partially known. Testing is still carried out by executing the implementation while knowledge about the structure of the implementation is used for test selection and adequacy.

An extra distinction made in the literature is between dynamic and static testing [KFN99; YT89]. In dynamic testing, an implementation is tested when executed. Contrary to dynamic testing, in static testing, an implementation is statically examined without being executed. This type of testing includes code inspection and some forms of analysis like requirement analysis. Note that this classification is orthogonal to the white/black-box testing. For instance, walkthroughs or symbolic execution are considered as static white-box testing. From the viewpoint of black-box testing, static testing can include reviewing requirements and specifications [HHT96]. However, some recent methods such as concolic testing [SMA05; Wil10] and runtime verification [LS09] have blurred the distinction between dynamic and static testing [Gau11]. In the literature, testing often refers to dynamic testing and static testing is often considered to be a verification activity. We follow this convention in this thesis as well.

1.3 Model-Based Testing

Traditionally, software testing is an expensive part of software development. It is common knowledge that testing consumes approximately 50 percent of the development time, and more than 50 percent of the total cost [MS04; EJ09; Tas02]. Considering the complexity
Chapter 1. Introduction

of software systems, manual testing is a laborious, time-consuming and error-prone activity in the development of software. Therefore in recent decades, test automation has received considerable attention. In many test automation projects in the industry, of all test activities, only automation of test execution is considered [GF12]. As observed in [LW04], automated execution of test cases does not solve the costly problem of testing by itself: automation of test case generation and a solution to the test oracle problem (which defines the result of testing) are required, too.

To automate test case generation and test oracle, a specification of the system has to be expressed in formal languages which are amenable to automated analysis. Tests are then automatically derived from those formal models, and subsequently executed. This approach of testing can be traced back to the seventies [Cho78] and is known as model-based testing in the literature. The recent focus on model-driven development, along with the maturity of formal verification tools and methods, has led to increased attention to this subject. Model-based testing can be exploited at any level of testing [Dal+99] and for a wide range of systems with rather different characteristics. In recent years, therefore, many different testing theories, methods and model-based testing tools have been developed both in the industry and academia. A general overview of different model-based testing approaches with respect to formal languages used to specify the behavior of systems, test generation algorithms and criteria, and policies of test execution are presented in recent surveys by Hierons et al. [Hie+09], Dias-Neto et al. [Dia+07] and Utting et al. [UPL12]. Besides all varieties, we can identify a generic process for model-based testing approaches: Building a formal specification of a system, choosing test selection criteria, generating and executing test cases, and delivering the final verdict of testing.

The key ingredient in model-based testing is a formal model that is used to describe a specification of the system. Specifications are abstractions representing particular aspects of the desired behavior of systems. While some details of systems are abstracted in specifications, they have to contain enough details to enable the derivation of both test generation and test oracle. Many different formal specification languages have been used with respect to a desired aspect of systems being tested. Hierons et al. [Hie+09] discusses many of these formal notations in detail, together with some examples of their supporting testing tools. Some of the most popular notations used in model-based testing can be generally categorized into the two following groups:

- State-based specifications: these model a system by describing the states of a system as a collection of variables together with invariants and operations governing their valuation. Examples of these notations include Z, VDM, B, and the C#-plus-preconditions of Spec Explorer [Vea+08].

- Action-based specifications: instead of describing admissible states of a system, these models characterize the transitions between states. Specifications described in finite state machines (FSMs), labeled transition systems (LTSs), I/O automata, and process algebras such as CSP [Hoa85], CCS [Mil89], and LOTOS [ISO89] are typical examples of this group. Testing such specifications is based on the definition of conformance described in terms of admissible input-output sequences.

More detailed classifications of formal modeling notations used to describe the specifications of systems are presented in [Hie+09; UPL12].
This thesis focuses on conformance testing of reactive systems, which continuously engage in interactions with their environment to accomplish a task. They receive stimuli triggered by their environment and produce outputs in response. Reactive systems are often modeled as a set of communicating processes. In this thesis, specifications of reactive systems are given as labeled transition systems which have a rich theory for modeling the behavior of concurrent processes and their communication [Ace+07; GM14].

1.4 Formal Conformance Testing

Conformance testing evaluates the functional behavior of an implementation with respect to its specification. Using the principles in ISO/IEC 9646 Standard [ISO94], ‘Formal Methods in Conformance Testing’ [CFP96] presents an abstract framework for using formal specifications in conformance testing, which is independent of the formal notation and the test algorithm used. The framework defines the main requirements needed for formal conformance testing; how to formally define conformance of an implementation with respect to its formal specification, how to formally reason about testing concepts such as test execution, and how to use testing concepts to test for conformance. Implementations under test are software systems in the real world. They are thus not amenable to formal reasoning. Yet, to enable formal reasoning about conformance of an implementation, it is assumed that the desired aspects of the behavior of the implementation under test are expressible in terms of a formal model. This is a minimum assumption that is made in model-based testing about implementations, known as the testing hypothesis. There are two approaches in model-based testing to formally define the notion of conformance.

The first approach makes an intensional comparison between the behavioral model of the implementation and a given specification of the system. With respect to a notion of conformance, implementations can be categorized into two classes, implementations that conform to the specification and implementations that do not. Thus, the notion of conformance is a binary relation between specifications and implementations, known as the implementation relation in model-based testing terminology.

In the second approach, the conformance of an implementation is characterized as the result of executing a set of test cases, derived from a formal specification with respect to test requirements, against the implementation. Such a definition, which is based on testing and observations, is known as an extensional definition in the literature [Tre96c; Tre96b; Tre96a]. Therefore, an implementation is considered to be a conforming implementation of a formal specification, if and only if it is accepted by all test cases derived from that specification.

To formally assess the conformance of an implementation with a formal specification by means of testing the two notions of conformance defined in the above approaches are sometimes linked. Therefore, the two conforming sets of implementations resulting from the above definitions are equivalent. Figure 1.2 illustrates the generic process of formal conformance testing.
1.5 Research Questions

This thesis focuses on the input-output conformance (ioco) relation [Tre96c]. The ioco relation is a commonly used implementation relation for black-box formal conformance testing of reactive systems. However, the ioco relation has some limitations in testing concurrent systems. Such systems are composed of interacting components and often communicate asynchronously with their environment. The main objective of this thesis is to propose possible solutions to make the ioco relation suited for testing concurrent systems by improving theoretical aspects of the ioco testing framework.

In order to achieve this objective, the limitations of the ioco relation in testing concurrent systems need to be identified. This is done through a case study, reported on in this thesis. In this case study, model-based testing techniques are applied on a concurrent system in the payment industry. The problems and challenges observed in this case study lead to a number of research questions. Each of these question is discussed in the remainder of this section.

The first issue identified in the case study concerns the use of the asynchronous communication channels in testing. The nature of communication in the case study reported in this thesis is asynchronous; however the ioco testing theory was developed based on the assumption that a tester can always communicate with an implementation under test synchronously. The case study has shown that testing an asynchronous system with this assumption results in false fail verdicts: testing reports a failure while the implemented behavior is as expected. The intuitive solution to this problem (as mentioned in [TV92; Tre92]) is to take account of the context, in this case FIFO buffers, in testing. Applying this solution has a negative effect on testing because of the dramatic increase in the size of the state-space of the specification. The following question, therefore, arises:

**RQ₁**: Under which conditions do the (standard) ioco test cases of a specifica-
tion result in a meaningful verdict in testing implementations that communicate only via asynchronous interactions?

To tackle the asynchrony problem in conformance testing, various approaches in both practice and theory are taken in the literature. Each of those solutions has its own effects on testing; some of them may change the notion of conformance testing completely in the synchronous and the asynchronous settings. In [WW09; Wei09], a solution is presented to cater for the asynchrony in input-output conformance testing, in which a restricted subclass of labeled transition systems (LTS), namely, internal-choice LTSs is used. However, the class of LTSs used in the \textit{ioco} relation differs from the class of internal-choice LTSs. Before applying the solution proposed in [WW09; Wei09] to the \textit{ioco} testing framework, it is essential to investigate restrictions that are imposed on testing by that subclass of LTSs. Therefore, the second research question is formulated as follows:

\textbf{RQ$_2$}: What is the relation between the testing power of the standard \textit{ioco} relation and the testing power of the implementation relation using internal-choice LTSs?

Another problem observed in the case study is the size of the state-space of specifications, even in the absence of FIFO buffers for tackling the asynchrony problem. Due to the very complex and concurrent nature of the system under test, the state-space of the specification is very large. The large state-space has a very negative impact on the performance of testing. Since the whole functionality of the system under test is often obtained from the interactions between different components, the decomposability question naturally arises:

\textbf{RQ$_3$}: Can a large monolithic specification be broken into smaller models with respect to its building components such that the conformity of the whole system is deducible from the testing results of the smaller specifications?

The other issue identified in the case study is the test selection problem. Exhaustive testing of systems is mostly impractical. The reported case study is not an exception. Considering the time and cost limitations of a real testing project, having good test selection criteria is essential for efficient testing. This fact has motivated several studies in the literature conducted in this area [Gau95; SP14]. There are many different implementation relations in the literature with various characteristics. For some of them, but not for the \textit{ioco} relation, efficient algorithms have been developed to assess those relations between models. As a step towards test selection it is desired to have a better insight into the theoretical limitations of the \textit{ioco} relation. In this regard, the following question is formulated:

\textbf{RQ$_4$}: What is the complexity of assessing the \textit{ioco} relation between IOLTSs in general?

1.6 Outline and Origin of Chapters

The remainder of this thesis is structured as follows; the origin of each chapter and the relevant dependencies among them are given.
Chapter 1. Introduction

Chapter 2  This chapter provides the formal concepts and background that are relevant to the $\text{ioco}$ testing framework and other theories developed in this thesis. In this regard, the general concepts of labeled transition systems, e.g., communication between LTSs, together with the concepts of testing using IOLTSs such as the concepts of test cases and conformance notions are formally defined in this chapter.

Chapter 3  This chapter reports on a case study in applying model-based testing on a payment system, namely, an Electronic Fund Transfer (EFT) switch. The challenges and problems that are observed in this case study motivate the theoretical studies conducted throughout this thesis. This chapter is based on the following publication:


Chapter 4  This chapter addresses research question $RQ_2$. To investigate the effect of the solution proposed in [WW09; Wei09] on testing, a comprehensive comparison between the $\text{ioco}$ relation and the implementation relations using internal-choice LTSs is conducted. To this end, the testing power of each implementation relation is formalized. Based on that formal definition, it is investigated when the testing power of those relations coincide so that they can be interchangeably used in testing. It is shown that besides the limitation that internal-choice LTSs imposes on modeling, the power of testing induced by the corresponding implementation relations differs from that of the $\text{ioco}$ relation. This chapter is based on the following paper:


The results described in Section 4.2.3 extend the results of Section 3 of this paper.

Chapter 5  This chapter addresses research question $RQ_4$. This chapter presents the computational complexity of deciding the standard $\text{ioco}$ relation between IOLTSs. It is first shown that the problem of deciding the $\text{ioco}$ relation is PSPACE-COMPLETE. A polynomial time algorithm is then presented for a restricted setting, where specifications are deterministic IOLTSs. The above algorithm relies on the redefinition of the $\text{ioco}$ relation as a simulation-like relation. The obtained results for the $\text{ioco}$ relation are also adopted for the implementation relations using internal-choice IOLTSs, a variant of IOLTSs, which is investigated in Chapter 4. This chapter is based on the following paper:


The results of this paper are entirely contained in Chapter 5. The novel contribution of this Chapter is the extension of the results of the aforementioned paper for the implementation relations using internal-choice traces, in Section 5.4. For this section it is recommended that Sections 4.1 and 4.3 are read first.
Chapter 6  This chapter answers research question RQ \textsubscript{1}. This chapter studies the problem of asynchronous communication in the \textit{ioco} testing framework. To be sure that standard \textit{ioco} test cases derived from a specification can be utilized in testing using asynchronous interaction between a tester and an implementation, the class of test cases, the class of implementations, or the class of specification, or a combination thereof has to be adapted. First, the case in which the class of test cases is limited to the internal-choice test cases is studied. It is shown that the verdicts of testing for this subclass of test cases are the same in both the synchronous and the asynchronous settings. Afterwards, it is argued that this subclass of test cases results in an implementation relation, whose testing power is different from that of the \textit{ioco} relation in the asynchronous setting. Therefore, the necessary and sufficient conditions needed to be imposed on implementations or specifications are studied in the later sections of Chapter 6. This chapter subsumes the results of the following two papers:


The results of the first paper and the results of sections 5 and 6 of the second paper are entirely contained in Chapter 6. In both papers, the problem of asynchronous communication in the \textit{ioco} testing framework are studied when the class of test cases is limited to the internal-choice test cases and when the class of implementations are confined to the sub-class of IOTSs called delay-right-closed IOTSs.

The novel contribution of this chapter is the introduction of a sub-class of IOLTSs as specifications whose verdicts are the same in both the synchronous and the asynchronous settings with respect to the standard \textit{ioco} test cases. The definition of this class of specifications and the corresponding theorems are presented in Section 6.3.

Chapter 7  This chapter addresses research question RQ \textsubscript{3}. It is well-known that the \textit{ioco} relation has only a limited form of compositionality \cite{BRT03}. In this chapter, the property of decomposability of a specification with respect to the \textit{ioco} relation is studied. The decomposability property of specifications allows for the \textit{ioco} relation to be used in component-based software or product-line software, where it is assumed that some parts of the system under test are reused components which are well-known and tested (referred to as a platform). This property is formally defined in the context of the \textit{ioco} relation. The decomposability property guarantees that a monolithic specification of a whole system can be broken into smaller specifications such that testing against those specifications is sufficient to assess the correctness of the whole system consisting of the platform and the unknown component. Thus, there is no need to re-test the integrated system. Knowing that a specification is decomposable is not enough for testing. One may want to find a witness of the decomposability property. In this regard, the problem of extracting the specification of an unknown component from a monolithic specification of
Chapter 1. Introduction

a system is studied. It is subsequently investigated when the derived specification can be a witness for the decomposability of the specification of the whole system with respect to the given platform. This chapter is based on the following paper:


Chapter 8 This chapter summarizes the main contributions and the results of this thesis. This thesis develops theoretical foundation relating to some of the problems experienced in the case study reported in Chapter 3. The goal of developing these theories is to propose solutions to make the ioco relation suited for testing concurrent systems. The evaluation of practical applicability of these theories, however, is the subject of future research. In this regard, required algorithms in the first place need to be implemented and to be integrated in the existing test tools that are based on the ioco relation. Subsequently, the effectiveness of these theories can be evaluated by applying these algorithms in different case studies. Moreover, there is still more room for theoretical improvement of the ioco testing framework, some of which are discussed for future work in Chapter 8.

Appendices contain parts of the proofs of the technical propositions and lemmas from Chapters 6 and 7. Moreover, some of the UPPAAL models used for testing the case study reported in Chapter 3 are given.
Chapter 2

Preliminaries

In this thesis, we focus on model-based testing which is a promising technique in testing, especially in functional testing. The main aim of this kind of testing is to find incorrect implementations with respect to a given formal specification by executing a set of experiments. The essential ingredients for this technique apart from a formal specification are an actual implementation, test cases and a notion of conformance.

In this thesis we focus on model-based testing of reactive systems which continuously engage in interactions with their environment to accomplish a task. Reactive systems can be modeled in several ways, e.g., finite state machines (FSMs) [YL98] and labeled transition systems (LTSs) [Tre08]. In this thesis we mostly concentrate on the ioco testing theory introduced by Tretmans [Tre96c; Tre08] and a variation of ioco presented by Weigelhofer and Wotawa in [WW09; Wei09], in which variants of LTSs are used as the behavioral models of desired systems and implementations.

This chapter presents the formal notations and definitions presented in the literature which are essential to establish the results of this thesis.

Structure of the chapter. An overview of the relevant concepts related to LTSs is given in Section 2.1. Section 2.2 summarizes general concepts and definitions of test cases and test execution for LTSs. We finish this chapter by studying the ioco testing theory introduced by Tretmans along with the conformance relation, presented by Weigelhofer and Wotawa in [WW09; Wei09], in Section 2.3.

2.1 Labeled Transition Systems

In this section, we introduce the concepts relevant to LTS-based testing theory explored in this thesis.

The labeled transition system model assumes that reactive systems can be represented using a set of states and transitions, labeled with events or actions, between such states. A tester can observe the events leading to new states, but she cannot observe the states. Events that are internal to a system, i.e., unobservable to a tester or observer of the system, are modeled by the action $\tau$, henceforth referred to as the internal action.
Consider the two LTSs depicted in Figure 2.1. Both of them model a vending machine which sells tea. After receiving a coin, it either refunds the coin, or after that the button is pressed, delivers tea. The set of actions of both LTSs contains \{coin, tea, refund\}. The set of states of LTSs \(c_0\) and \(e_0\) is \{\{0\}, \{1\}, \{2\}, \{3\}\} and \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}\} respectively. We observe that in LTS \(e_0\), after receiving a coin, the system can internally move to state \(e_2\) where pressing the button is disabled.

A word is a sequence over actions. The set of all words over \(L\) is denoted by \(L^*\), and \(\epsilon\) is the empty word. For words \(\sigma, \rho \in L^*\), we denote the concatenation of \(\sigma\) and \(\rho\) by \(\sigma \rho\). The transition relation \(\rightarrow\) is generalized to a relation \(\rightarrow^* \subseteq S \times L^* \times S\) over a word by the following deduction rules, using traditional Plotkin-style rules [Plo04]:

\[
\begin{align*}
\frac{}{s \xrightarrow{\epsilon} s} \quad & s \xrightarrow{\sigma} s'' \quad s'' \xrightarrow{x} s' \quad x \neq \tau \quad & s \xrightarrow{\sigma^* s''} s'' \xrightarrow{\tau} s' \\
\end{align*}
\]

In line with our notation for transitions, we write \(s \xrightarrow{\sigma} s'\) if there is some \(s'\) such that \(s \xrightarrow{\sigma^* s'}\), and \(s \not\xrightarrow{\sigma^*}\) when not \(s \xrightarrow{\sigma^*}\).

An LTS \(\check{s}\) is said to be deterministic if the set of reachable states after executing any sequence of actions (a word) always has at most one element; that is, for all \(s, s', s'' \in S\) and all \(\sigma \in L^*\), if \(s \xrightarrow{\sigma^* s'}\) and \(s \xrightarrow{\sigma^* s''}\) then \(s' = s''\).

A state in the LTS \(\check{s}\) is said to diverge if it is the source of an infinite sequence of \(\tau\)-labeled transitions. The LTS \(\check{s}\) is divergent if one of its reachable states diverges. Throughout this thesis, we confine ourselves to non-divergent LTSs unless indicated otherwise.
2.1. Labeled Transition Systems

Example 2.3. Consider again the LTSs depicted in Figure 2.1. We observe that LTS $c_0$ is deterministic, since after executing any word $\sigma \in \{\text{coin, refund, button, tea}\}^*$ at a state $s \in \{c_0, c_1, c_2, c_3\}$, at most one state is reachable, e.g., from the initial state $c_0$ after executing sequence $\text{coin refund}$, only state $c_2$ is reachable, i.e., $c_0 \xrightarrow{\text{coin refund}} c_2$. Obviously, it is not the case in LTS $e_0$, because after executing the word $\text{coin}$, the system can end up either at state $e_1$ or $e_2$. LTS $e_0$, therefore, is non-deterministic, because of the internal transition $e_1 \xrightarrow{\tau} e_2$.

Let $s, s' \in S$. A word $\sigma \in L^*$ is said to be a trace of the state $s$ if $s \xrightarrow{\sigma}$. State $s'$ is, subsequently, said to be reachable from the state $s$ if there exists a trace leading to that state, i.e., $\exists \sigma \in L^* \cdot s \xrightarrow{\sigma} s'$. We formally define the set of traces, the set of reachable states and the set of (weakly) enabled actions of state $s$ in the following.

Definition 2.4 (Traces, reachable states, and enabled actions). Let $s \in S$ and $S' \subseteq S$. The set of traces, the set of reachable states, enabled actions and weakly enabled actions for $s$ and $S'$ are defined as follows:

1. $\text{traces}(s) = \{ \sigma \in L^* \mid s \xrightarrow{\sigma} \}$, and we define $\text{traces}(S') = \bigcup_{s \in S} \text{traces}(s)$

2. $\text{der}(s) = \{ s' \in S \mid \exists \sigma \in L^* \cdot s \xrightarrow{\sigma} s' \}$, and we define $\text{der}(S') = \bigcup_{s \in S} \text{der}(s)$

3. $\text{init}(s) = \{ a \in L \cup \{\tau\} \mid s \xrightarrow{a}\}$, and we define $\text{init}(S') = \bigcup_{s \in S} \text{init}(s)$,

4. $\text{Sinit}(s) = \{ a \in L \mid s \xrightarrow{a}\}$, and we define $\text{Sinit}(S') = \bigcup_{s \in S} \text{Sinit}(s)$.

2.1.1 Communication Models

A system or a module usually communicates with others in an orchestrated fashion to accomplish a task or achieve a desired behavior. Such a communication can be synchronously or asynchronously carried out. In synchronous communication, systems do communicate directly with each other, while in the latter form, they communicate via channels with some (unspecified) delay. These can be modeled as unbounded buffers, see Figure 2.2.

![Figure 2.2: A schematic view of communication between two systems; (a) a synchronous communication, (b) an asynchronous communication](image-url)
Chapter 2. Preliminaries

Synchronous communication. In our setting, we formalize synchronous communication using the parallel composition operator $\parallel : \text{LTS} \times \text{LTS} \rightarrow \text{LTS}$, formally defined below: two LTSs can interact by synchronizing on their common actions. For the non-common actions the behavior of the LTSs is interleaved.

**Definition 2.5** (parallel composition). Let $\langle S_1, L_1, \rightarrow_1, \bar{s}_1 \rangle$ and $\langle S_2, L_2, \rightarrow_2, \bar{s}_2 \rangle$ be two LTSs. The parallel composition of $\bar{s}_1$ and $\bar{s}_2$, denoted by $\bar{s}_1 \parallel \bar{s}_2$, is the LTS $\langle Q, L, \rightarrow, \bar{s}_1 \parallel \bar{s}_2 \rangle$, where:

- $Q = \{ s_1 \parallel s_2 \mid s_1 \in S_1, s_2 \in S_2 \}$.
- $L = L_1 \cup L_2$
- $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ is the least relation satisfying:

\[
\begin{align*}
& s_1 \xrightarrow{x} s_1' \quad x \not\in L_2 \quad s_1 \parallel s_2 \xrightarrow{x} s_1' \parallel s_2 \quad s_2 \xrightarrow{x} s_2' \quad x \not\in L_1 \quad s_1 \parallel s_2 \xrightarrow{x} s_1' \parallel s_2' \\
& s_1 \parallel s_2 \xrightarrow{x} s_1' \parallel s_2' \quad x \neq \tau \quad s_1 \parallel s_2 \xrightarrow{x} s_1' \parallel s_2'
\end{align*}
\]

**Example 2.6.** Consider the LTSs depicted in Figure 2.3. The LTS $\bar{s}$ is a specification of a vending machine which sells drinks. After receiving a coin, it either immediately delivers tea or serves coffee after its sugar button is pressed. The LTS $\bar{c}$ presents the behavioral model of a tea-obsessive computer scientist who always likes to drink tea (with or without sugar). Let our computer scientist who behaves as LTS $\bar{c}$ use a vending machine the behavioral model of which is $\bar{s}$. Their communication can be modeled by the parallel composition of the LTSs $\bar{s}$ and $\bar{c}$ where their common actions is $\{\text{coin}, \text{black-tea}, \text{sugar}\}$.

At the beginning, the computer scientist inserts a coin and the machine at the same time receives it and proceeds to the next states, i.e., $\bar{s} \parallel \bar{c} \xrightarrow{\text{coin}} s_1 \parallel c_1$. Now, either the machine delivers tea and the computer scientist takes it, i.e., $s_1 \parallel c_1 \xrightarrow{\text{black-tea}} s_2 \parallel c_2$ or she adds sugar to her cup by pressing the sugar button, i.e., $s_1 \parallel c_1 \xrightarrow{\text{sugar}} s_3 \parallel c_3$. Upon the sugar button being pressed, the machine delivers coffee while the computer scientist does not take it and waits for tea in vain, i.e., $s_3 \parallel c_3 \xrightarrow{\text{coffee}} s_4 \parallel c_4$.

Asynchronous communication. We now turn our attention to asynchronous communication. A system interacts asynchronously with another one by reading from and writing into buffers. Unlike synchronous communications where interactions between systems are modeled as a single step, in asynchronous setting, communication is carried out in two steps: writing in a buffer at one end and reading from that buffer at the other end. Therefore, from the view point of each system, there are two different types of buffers; sending and receiving buffers, Figure 2.2(b). An asynchronous channel in general can be the source of delays, message reordering or message loss in a communication. In this thesis, we study the asynchronous channels which cause only delays in communication. Such channels can be modeled by reliable FIFO queues. A FIFO queue models delays occurring in asynchronous communication while it preserves the order of messages. We
2.1. Labeled Transition Systems

Figure 2.3: Two LTS models where (a) pictures the behavioral model of a vending machine and (b) depicts the behavioral of a tea-obsessive computer scientist

usually model an asynchronous system as a synchronous one by hiding the interactions of one of the systems with the buffers. In this vein, four different types of scheme can be assumed which are comprehensively studied in [Beo12]. We here present only one of the models in [Beo12] that is essential for testing.

Figure 2.4: A synchronized schematic view of asynchronous communication

By adding buffers to one side, we synchronize the asynchronous communication, Figure 2.4. Therefore, the interactions of that system with buffers are considered as the internal transitions and are obviously hidden from the viewpoint of external observer. By adding channels to a system, its visible behavior changes. This is formalized below.

Definition 2.7 (Queue operator). Let \( \langle S, L, \rightarrow, \bar{s} \rangle \) be an arbitrary LTS, and let \( \sigma_s, \sigma_r \in L^* \) denote actions that are already in the send and the receive buffers respectively. The unary queue operator \( [\sigma_s \ll s \ll \sigma_r] \) is then defined by the following axioms and inference rules:

- **A1**: \( [\sigma_s \ll s \ll \sigma_r] \xrightarrow{a} [\sigma_s \ll s \ll \sigma_r, a] \)
- **A2**: \( [\sigma_s \ll s \ll \sigma_r] \xrightarrow{\tau} [\sigma_s \ll s' \ll \sigma_r] \)
- **I1**: \( s \xrightarrow{\tau} s' \xrightarrow{\tau} [\sigma_s \ll s' \ll \sigma_r] \)
- **I2**: \( s \xrightarrow{a} s' \xrightarrow{\tau} a \in L \xrightarrow{\tau} [\sigma_s \ll s' \ll \sigma_r] \)
- **I3**: \( s \xrightarrow{a} s' \xrightarrow{\tau} a \in L \xrightarrow{\tau} [\sigma_s \ll s' \ll \sigma_r] \)
We abbreviate \([e \ll s \ll \varepsilon]\) to \(Q(s)\). We observe that \(Q(s)\) defines again an LTS. We have the following property, relating the traces of an LTS to the traces it has in the queue context [Nor+14].

**Property 2.8.** Let \((S, L, \rightarrow, s)\) be an arbitrary LTS. Then for all \(s, s' \in S\), we have \(s \overset{\sigma}{\Rightarrow} s'\) implies \(Q(s) \Rightarrow Q(s')\).

We now formalize the asynchronous communication between LTSs by using the parallel composition operator. In this regard, we define the asynchronous composition of two LTSs \(\tilde{s}_1\) and \(\tilde{s}_2\), where LTS \(\tilde{s}_2\) is considered as an observer, as an LTS that behaves like \(Q(\tilde{s}_1)\parallel \tilde{s}_2\).

**Example 2.9.** Consider the LTSs \(\tilde{s}\) and \(\tilde{c}\) depicted in Figure 2.3. Let a computer scientist, this time, interact with the vending machine asynchronously, for example via the web interface of the machine. At the beginning, she sends a coin and presses the sugar button successively which are put in the receive buffer of the machine, i.e., \(Q(\tilde{s})\parallel \tilde{c} \xrightarrow{\text{coin sugar}} \tilde{c}_3\). The machine, then, picks and processes the received actions from its receive buffers. Upon processing the action \(\text{coin}\), the vending machine may deliver and put \(\text{tea}\) via an internal transition in its send buffer, i.e., \([e \ll \tilde{s} \ll \text{coin sugar}]\parallel \tilde{c}_3 \xrightarrow{\varepsilon} [\text{black-tea} \ll \tilde{s}_2 \ll \text{sugar}]\parallel \tilde{c}_3\). Although the computer scientist has requested tea with sugar, black tea is delivered by the machine. Regarding Definition 2.5, the system obtained from the asynchronous composition of LTSs \(\tilde{c}\) and \(Q(\tilde{s})\) cannot proceed any further from the state \([\text{black-tea} \ll \tilde{s}_2 \ll \text{sugar}]\parallel \tilde{c}_3\). Note that at the state \([\text{black-tea} \ll \tilde{s}_2 \ll \text{sugar}]\parallel \tilde{c}_3\), the computer scientist is doing nothing but waiting for receiving tea with sugar while the machine delivered tea.

### 2.1.2 Specification Models

Reactive systems react to the stimuli received from their environment by changing their state or producing and sending out a response in turn. Although in LTSs labels are treated uniformly, when engaging in an interaction with another system, the actions of an LTS can be assumed to be partitioned into two subcategories: input and output actions. Reflecting which of the systems takes the initiative in executing the action, input actions are under the control of the environment of the system, whereas output actions are under the control of the system itself. We refine the LTS model to reflect this distinction in initiative.

**Definition 2.10** (IOLTS). Let \(I\) be the input actions, and let \(U\) be the output actions such that \(I \cap U = \emptyset\). An input-output labeled transition system (IOLTS) is a 5-tuple \((S, I, U, \rightarrow, \tilde{s})\) such that the tuple \((S, L, \rightarrow, \tilde{s})\) with \(L = I \cup U\) is an LTS.

We denote the class of IOLTS models ranging over \(I\) and \(U\) by IOLTS\((I, U)\).

Throughout this thesis, whenever we are dealing with an IOLTS (or one of its refinements), the alphabet of a given IOLTS is denoted by \(L\), which is partitioned into sets \(I\) and \(U\), i.e., \(L = I \cup U\). In the literature, inputs are sometimes distinguished from outputs by annotating them with a question-mark (?) and exclamation-mark (!), respectively. Note that these annotations are not part of action names.

As stated in Section 2.1.1, asynchronous communication of an LTS with another one is carried out by writing to and reading from the send and receive buffers, respectively: the actions initiated by the LTS are written in the send buffers, and the actions provided by
2.1. Labeled Transition Systems

its counterpart are picked from the receive buffer. By distinguishing inputs and outputs in IOLTSs, we observe that in the asynchronous setting, an IOLTS writes its outputs in the send buffer and reads its inputs from the receive buffer. We refine the definition of the queue operator, given below, to reflect this distinction [Tre92].

**Definition 2.11 (Queue operator for IOLTSs).** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an arbitrary IOLTS, and let \( \sigma_s, \sigma_r \in L^* \) denote actions that are already in the send and the receive buffers respectively. The unary queue operator \( \sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} \sigma_r \) is then defined by the following axioms and inference rules:

\[
\begin{align*}
A1 & : & a \in I & \quad & \sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} \sigma_r \rightarrow [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} a] \\
I1 & : & s \xrightarrow{\tau} s' & \quad & [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} \sigma_r] \rightarrow [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} a] \\
A2 & : & a \in U & \quad & a \ll_\mathcal{Q} \ll_\mathcal{Q} \sigma_r \rightarrow [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} a] \\
I2 & : & s \xrightarrow{\tau} s' & \quad & [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} \sigma_r] \rightarrow [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} a] \\
I3 & : & s \xrightarrow{\tau} s' & \quad & a \in I & \quad & [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} a] \rightarrow [\sigma_s \ll_\mathcal{Q} \ll_\mathcal{Q} a]
\end{align*}
\]

Observations of output, and the absence thereof, are essential ingredients in the conformance testing theories we consider. A system state that does not produce outputs is called quiescent. In its traditional phrasing, quiescence characterizes system states that do not produce outputs and which are stable, i.e., those that cannot evolve to another state by performing a silent action.

**Definition 2.12 (Quiescence).** State \( s \in S \) is called quiescent, denoted by \( \delta(s) \), iff \( \text{init}(s) \subseteq I \). We say \( s \) is weakly quiescent, denoted by \( \delta_q(s) \), iff \( \text{Sinit}(s) \subseteq I \).

The notion of weak quiescence is appropriate in the asynchronous setting, where the lags in the communication media interfere with the observation of quiescence: an observer cannot tell whether a system is engaged in some internal transitions or has come to a standstill. By the same token, in an asynchronous setting it becomes impossible to distinguish divergence from quiescence; we re-visit this issue in chapter 6, asynchronous input-output conformance testing. Furthermore, in line with the definition of inputs and outputs, we refer to the send and receive buffers as the input and output buffers respectively. More precisely, interacting asynchronously with its environment, a system puts its outputs in the output buffer and similarly, gets the inputs from the input buffer.

In order to formally reason about the observations of inputs, outputs and quiescence, we introduce the set of suspension traces. Let \( L_\delta \) denote the set \( L \cup \{ \delta \} \), where \( L = I \cup U \). We first generalize the transition relation \( \rightarrow^* \) to a relation \( \Rightarrow \subseteq S \times L_\delta^* \times S \) over a sequence of input, output and quiescence actions.

\[
\begin{align*}
& s \xrightarrow{\sigma} s' & \xrightarrow{\delta} & s' & \xrightarrow{\rho} s' \\
& s \Rightarrow s' & \Rightarrow s & \Rightarrow s' \\
& s \xrightarrow{\sigma \rho} s' & \xrightarrow{\rho} s' & \xrightarrow{\rho} s' \\
& s \xrightarrow{\sigma \rho} s' & \xrightarrow{\rho} s' & \xrightarrow{\rho} s'
\end{align*}
\]
Example 2.13. Consider the LTS $c_0$ in Figure 2.1(a). The coin-labeled transition in LTS $s$ is triggered once a coin is provided by the environment; thus coin is an input of LTS $c_0$. Subsequently, in response, LTS $c_0$ may deliver tea via performing the tea-labeled transition; thus, tea is considered as an output of LTS $c_0$. In this vein, the language of IOLTS $c_0$ can be partitioned into two disjoint sets $\{\text{coin}, \text{button}\}$ as inputs and $\{\text{tea}, \text{refund}\}$ as outputs. IOLTS $c_0$ at its initial state is stable and does not produce any output either. Thus, IOLTS $c_0$ is quiescence at its initial state, i.e., it holds that $\delta(c_0)$.

The following definition formalizes the set of suspension traces.

Definition 2.14. Let $s \in S$ and $S' \subseteq S$. The set of suspension traces for $s$, denoted by $\text{Straces}(s)$ is defined as the set $\{\sigma \in L^*_s \mid s \xrightarrow{\sigma} \}$; we set $\text{Straces}(S') = \bigcup_{s' \in S'} \text{Straces}(s')$.

We next recall the specialization of IOLTSs, introduced in [WW09] by Weiglhofer and Wotawa. We study testing theories using this subclass of IOLTSs in more details in chapters 4 and 6.

Definition 2.15 (Internal-choice IOLTS). An IOLTS $\langle S, I, U, \rightarrow, \bar{s} \rangle$ is an internal choice input-output labeled transition system (IOLTS$^\cap$), iff only quiescent states may accept inputs, i.e., for all $s \in S$, if $\text{init}(s) \cap I \neq \emptyset$ then $\delta(s)$.

We denote the class of IOLTS$^\cap$ models ranging over $I$ and $U$ by IOLTS$^\cap(I, U)$. An IOLTS$^\cap$ model represents the behavior of a system in which the input actions can be accepted only when the system is stable, i.e., at quiescent states. This restriction, however, does not force inputs and outputs to be necessarily interleaved. Therefore, by using internal-choice IOLTSs, we still remain in the realm of LTS testing theories and we are not back at FSM testing, in which inputs and outputs are assumed to alternate.

Example 2.16. Consider four IOLTSs $c_0$, $e_0$, $o_0$ and $i_0$ in Figure 2.5. All of them model a vending machine which, after receiving coin, either refunds it, or after that the button is pressed, produces tea. In IOLTS $c_0$, after receiving coin, there is a choice between input and output; the exact behavior modeled by the transition system is, arguably, awkward, as by pressing a button coin refund can be prevented. Although IOLTS $e_0$ does not feature an immediate race between input and output actions, the possibility of output refund can still be ruled out by providing input button. IOLTS $o_0$ in Figure 2.5 models a malfunctioning vending machine which, after pressing the button, may or may not deliver tea. IOLTS $i_0$ does not contain this fault and can be considered a reasonable specification of a vending machine. Neither $c_0$, nor $e_0$ belongs to the class IOLTS$^\cap$, whereas $o_0$ and $i_0$ do. Namely, in the two IOLTSs $o_0$ and $i_0$, input actions, i.e., $\{\text{coin, button}\}$, are only enabled in states where no output or internal action is enabled.

2.1.3 Implementation Models

Several formal testing theories build on the assumption that the implementations can be modeled by a particular IOLTS; this assumption is part of the so-called testing hypothesis underlying the testing theory. Not all theories rely on the same assumptions. We introduce two models, viz., the input-output transition systems, used in Tretmans’ testing theory.
2.1. Labeled Transition Systems

Figure 2.5: IOLTSs with different moments of choice

[Tre96c; Tre08] and the internal-choice input-output transition systems, introduced by Weiglhofer and Wotawa [WW09].

Tretmans’ input-output transition systems, formally defined below, are basically plain IOLTSs with the additional assumption that inputs can always be accepted.

**Definition 2.17** (IOTS). Let \( \langle S, I, \rightarrow, \bar{s} \rangle \) be an IOLTS. A state \( s \in S \) is **input enabled** iff \( I \subseteq S_{\text{init}}(s) \); the IOLTS \( \bar{s} \) is an **input-output transition system** (IOTS) iff every state \( s \in S \) is input enabled.

The class of input-output transition systems ranging over inputs \( I \) and outputs \( U \) is denoted by \( \text{IOTS}(I, U) \).

The input-enabledness of an implementation ensures that it never refuses stimuli triggered by its environment. In other words, regardless of the state of such an implementation, a tester can always stimulate the implementation during test execution. While the ioco testing theory assumes input-enabled implementations, it does not impose this requirement on specifications. This facilitates testing using partial specifications, i.e., specifications that are under-specified. Therefore, it allows for specifying only that part of the behavior of the system-under-test that is relevant to testing.

**Example 2.18.** Consider the IOLTSs depicted in Figure 2.5, where the set of their inputs is \( \{ \text{coin, button} \} \). IOLTS \( c_0 \) is not input enabled, and neither is \( e_0 \): for example, after input \( \text{coin} \), neither of the two allow for input \( \text{coin} \) any more. IOLTS \( o_0 \) is not input enabled either, because for example at state \( o_5 \) it refuses to accept any input. The aforementioned IOLTSs can be made IOTSs by adding self-loops for all absent input transitions at each and every state. IOLTS \( i_0 \) is input enabled, however, and is thus an IOTS.
Weiglhofer and Wotawa’s internal-choice input-output transition systems relax Tretmans’ input-enabledness requirement; at the same time, however, they impose an additional restriction on the presence of inputs, which stems from the fact that their class of implementations specialize the IOLTS class.

**Definition 2.19 (Internal-choice IOTS).** An IOLTS\(^\cap\) \((S, I, U, \rightarrow, \bar{s})\) is an internal choice input-output transition system (IOLTS\(^\cap\)), iff every quiescent state is input enabled, i.e., for all \(s \in S\), if \(\delta(s)\), then \(I \subseteq S_{init}(s)\).

We denote the class of IOLTS\(^\cap\) models ranging over \(I\) and \(U\) by IOLTS\(^\cap\)(\(I, U\)). The following Venn-diagram depicts the relation between the IOLTS, IOLTS\(^\cap\), IOTS and IOTS\(^\cap\) models.

![Venn-diagram](image)

**Example 2.20.** Consider IOLTSs depicted in Figure 2.5. Both \(o_0\) and \(i_0\) belong to the class IOLTS\(^\cap\). IOLTS\(^\cap\) \(i_0\) is input enabled and hence is also an IOLTS\(^\cap\). IOLTS\(^\cap\) \(o_0\) is input-enabled in all states but \(o_4\) and \(o_5\), and since these two states are not quiescent, it follows from Definition 2.19 that \(o_0\) is indeed an IOLTS\(^\cap\).

Consider IOLTSs pictured in Figure 2.6. Both IOLTSs \(\tilde{r}\) and \(\tilde{i}\) are categorized into IOTS class because all input actions, in this case \(coin\), are enabled at every single state via possible \(\tau\)-labeled transitions. IOTS \(\tilde{i}\) is an IOTS\(^\cap\) as well, because it accepts \(coin\) as the only input action only at quiescent states. Opposite to IOTS \(\tilde{i}\), an input action \(coin\), in IOTS \(\tilde{r}\) is enabled at non-quiescent state, e.g., at state \(r_1\), IOTS \(\tilde{r}\) may produce tea as an output or accept \(coin\) as an input. Thus, IOTS \(\tilde{i}\) is not an IOTS\(^\cap\).

![Two input-enabled IOLTS models of a vending machine](image)

**2.1.4 Suspension Automaton**

In this thesis, many of the presented testing theories rely on an automaton derived from an IOLTS specification. This automaton, called a suspension automaton, shares many of
2.1. Labeled Transition Systems

Figure 2.7: Suspension automata modeling a vending machine; two valid suspension automata $\bar{q}$ and $\bar{p}$ and an invalid suspension automaton $\bar{o}$

the characteristics of an IOLTS, except that the observations of quiescence are encoded explicitly as outputs: $\delta$ is treated as an ordinary action label which can appear on a transition. In addition, these suspension automata are assumed to be deterministic: any word that could be produced by an automaton leads to exactly one state in the automaton.

**Definition 2.21** (Suspension automaton). A suspension automaton (SA) is a deterministic LTS $\langle S, L \cup \{\delta\}, \rightarrow, \bar{s} \rangle$ with $L = I \cup U$; that is, for all $s \in S$ and all $\sigma \in L^*$, we have the number of reachable states after executing $\sigma$ from state $s$ is at most one.

**Example 2.22.** Consider LTSs depicted in Figure 2.7. These are three suspension automata over the language $L_\delta$ where $L = I \cup U$ with $I = \{\text{coin}\}$, and $U = \{\text{tea, refund}\}$.

Note that determinism implies the absence of $\tau$ transitions. In [Tre96c], a transformation from ordinary IOLTSs to suspension automata is presented. The transformation, given below, can be considered as adding a $\delta$ self-loop at quiescent states and then determinizing the obtained automaton.

**Definition 2.23** (Transformation operator $\Delta$ [Tre96c]). Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOLTS. The obtained suspension automaton from $\bar{s}$, denoted by $\Delta(\bar{s})$, is an LTS $\langle Q, L \cup \{\delta\}, \rightarrow_q, \bar{q} \rangle$ with $L = I \cup U$, where:

- $Q = P(S) \setminus \emptyset$, where $P(S)$ is the power set of $S$
- $\bar{q} = \{s \mid \bar{s} \xRightarrow{\epsilon} s\}$
- $\rightarrow_q \subseteq Q \times L_\delta \times Q$ is the least relation satisfying:

$$
\begin{align*}
\text{for } & a \in L, s \in q, q' \in Q & \text{we have } s \xrightarrow{a} q' \Rightarrow & (s, q, q') \\
\text{for } & \delta(s) \in L_\delta & \text{we have } q \xrightarrow{\delta} q' \Rightarrow & (s, q, \delta(s))
\end{align*}
$$

The transformation ensures that the set of traces of the resulting suspension automaton is exactly as the set of suspension traces of the original IOLTS.

**Proposition 2.24** (Proposition in [Tre96c]). Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOLTS. Then, we have traces$(\Delta(\bar{s})) = \text{Straces}(\bar{s})$. 

23
Though every IOLTS is transformable to a suspension automaton, there is an infinite number of suspension automata which do not represent any specification modeled by IOLTSs. In [Wil06], the set of suspension automata is characterized for which a transformation to an IOLTS is possible. Such suspension automata, defined below, are called valid.

**Definition 2.25** (Valid suspension automaton). Let \( \langle Q, L \cup \{\delta\}, \rightarrow, \bar{q} \rangle \) with \( L = I \cup U \) be a suspension automaton. We say that \( \bar{q} \) is valid iff it satisfies the following conditions:

- (non-blocking) for all \( q \in \text{der}(\bar{q}) \), if \( \text{init}(q) \cap U = \emptyset \), then \( \delta \in \text{init}(q) \).
- (quiescent reducible) for all \( q \in \text{der}(\bar{q}) \), \( \sigma \in L^* \), if \( \delta \sigma \in \text{traces}(q) \) then \( \sigma \in \text{traces}(q) \).
- (anomaly-free) for all \( q \in \text{der}(\bar{q}) \) and \( x \in U \), we have \( \delta x \notin \text{traces}(q) \).
- (stable) for all \( q \in \text{der}(\bar{q}) \), there is some set \( \Sigma \subseteq L^*_\delta \) such that for all \( \sigma \in \delta^* \) and all \( q' \in Q \), if \( q \xrightarrow{\delta \sigma} q' \) then \( \text{traces}(q') = \Sigma \).

**Example 2.26.** Consider IOLTSs and suspension automata pictured in Figures 2.6 and 2.7, respectively. Suspension automaton \( \bar{q} \) is actually obtained from IOLTS \( \bar{r} \) according to Definition 2.23. It is observed that the set of traces of \( \bar{q} \) equals to the set of suspension traces of \( \bar{r} \) as it is expected according to Proposition 2.24. Since Suspension automaton \( \bar{q} \) is non-blocking, quiescent reducible, anomaly-free and stable, it is a valid suspension automaton. Similarly, suspension automaton \( \bar{p} \) which is derived from IOLTS \( \bar{i} \) by applying the transformation algorithm in Definition 2.23, is valid. But, suspension automaton \( \bar{o} \) represents an invalid one because it violates the anomaly-free properties, i.e., after observing quiescence at state \( \bar{o} \), output actions tea and refund are enabled. Due to definition of quiescence in IOLTSs, this situation cannot be modeled by any IOLTS model.

The proposition below states that \( \Delta(\bar{s}) \) is a valid suspension automaton.

**Proposition 2.27** (Proposition [Wil06]). Let \( \bar{s} \) be an IOLTS. Then, \( \Delta(\bar{s}) \) is valid.

### 2.2 Test Cases

In testing a set of experiments are run on the implementation under test to assess if its behavior conforms to a given specification. These experiments are specified in test cases. A test case, in the most general case, can be described by a tree-shaped IOLTS. Such a test case prescribes when to stimulate an implementation under test (henceforth, referred to as IUT) by sending an input, and when to observe outputs emitted by the IUT. In general, the inputs to a test case are the outputs of the IUT, whereas the outputs of a test case are the inputs of the IUT. In order to formally distinguish between observing quiescence and “being” quiescent, a special action label \( \theta \) is introduced, which stands for the former. Since we sometimes reason about the behavior \( \sigma \) of an implementation from the viewpoint of a tester, we interpret \( \delta \) labels as \( \theta \) labels; formally, we then write \( \overline{\sigma} \) to denote the sequence \( \sigma \) in which all \( \delta \) labels have been replaced by \( \theta \) labels.

**Definition 2.28** (Test case). A test case is an IOLTS \( \langle S, I, U, \rightarrow, \bar{s} \rangle \), in which:
2.2. Test Cases

1. $S$ is a finite set of states reachable from $\bar{s}$,
2. terminal nodes of $S$ are called pass or fail, i.e., $\text{pass}, \text{fail} \in S$ and $\init(\text{pass}) = \init(\text{fail}) = \emptyset$
3. the quiescence observation $\theta$ belongs to $I$, i.e., $\theta \in I$
4. the transition relation $\rightarrow$ is acyclic, and deterministic.
5. pass and fail states appear only as targets of transitions labeled by an element of $I$, and
6. for all non-terminal states $s$, either $\init(s) = I$ or $\init(s) = (I \setminus \{\theta\}) \cup \{x\}$ for some $x \in U$.

We denote the class of test cases ranging over inputs $I$ and outputs $U$ by $TTS(U, I)$.

We next formalize what it means to execute a test case on an IUT. The intuition is that whenever a test case stimulates the IUT by sending an input, the latter consumes the input and responds by moving to a (possibly new) next state. In the same vein, whenever the IUT issues an output, the output is consumed by the test case, upon which the test case moves to a next state. A test case is considered successfully executable on an implementation if and only if in any communication with that implementation, it always reaches to a terminal state. Observe that the communication between the test case and the IUT can be instantaneous (i.e., synchronous), or through some underlying infrastructure that may introduce delays in the communication (i.e., communication is asynchronous). In most of the testing theories including the ones that are studied in this thesis, communication between implementations and test cases is assumed to be synchronous, formally defined below.

**Definition 2.29** (Synchronous execution). Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOLTS, and let $\langle T, U \cup \{\theta\}, I, \rightarrow, \bar{t} \rangle$ be a test case. Let $s, s' \in S$ and $t, t' \in T$. Then the synchronous execution of the test case $\bar{t}$ and $\bar{s}$, denoted by $\bar{t} \parallel s$, is defined through the following inference rules:

$$\frac{s \overset{\tau}{\rightarrow} s'}{t \parallel s \overset{\tau}{\rightarrow} t \parallel s'} \quad R1$$

$$\frac{t \overset{\bar{x}}{\rightarrow} t' \quad s \overset{\bar{x}}{\rightarrow} s'}{t \parallel s \overset{\bar{x}}{\rightarrow} t' \parallel s'} \quad R2$$

$$\frac{t \overset{\theta}{\rightarrow} t' \quad \delta(s)}{t \parallel s \overset{\theta}{\rightarrow} t' \parallel s} \quad R3$$

Following the definition above, we have the property below which extends the test synchronous execution over the internal behaviors of an implementation [Nor+14].

**Property 2.30.** Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOLTS, and let $\langle T, U \cup \{\theta\}, I, \rightarrow, \bar{t} \rangle$ be a test case. Let $s, s' \in S$ and $t, t' \in T$. Then, $s \overset{\varepsilon}{\rightarrow} s'$ iff $t \parallel s \overset{\varepsilon}{\rightarrow} t \parallel s'$.

Analogous to the asynchronous communication between LTSs, we can synchronize the asynchronous test execution. To this end, the scope of the IUT is extended and the asynchronous channels are considered as a part of the implementation. Test cases, now, can be synchronously executed against the extended implementation. Furthermore, the remote observer (asynchronous tester) cannot distinguish whether the system under test is stable or is engaged in some internal transitions; thus, weak quiescence is adopted to denote quiescence in the queue context.
Definition 2.31 (Synchronous execution in the queue context). Let \( \langle S, I, U, \rightarrow_s, \bar{s} \rangle \) be an IOLTS, and let \( \langle T, U \cup \{ \theta \}, I, \rightarrow_t, \bar{t} \rangle \) be a test case. Let \( s, s' \in S \) and \( t, t' \in T \). Then the synchronous execution of the test case and \( Q(\bar{s}) \) is defined through the following inference rules:

\[
\begin{align*}
\frac{[\sigma_s \triangleleft s \triangleleft \sigma_i]}{t \mid [\sigma_s \triangleleft s \triangleleft \sigma_i] \xrightarrow{\bar{s}} t \mid [\sigma_s' \triangleleft s' \triangleleft \sigma_i']} & \quad \text{R1'} \\
\frac{t \xrightarrow{\bar{t}} t'}{[\sigma_s \triangleleft s \triangleleft \sigma_i] \xrightarrow{\bar{t}} [\sigma_s' \triangleleft s' \triangleleft \sigma_i']} & \quad \text{R2'} \\
\frac{t \xrightarrow{\bar{t}} t' \quad \delta_q([\sigma_s \triangleleft s \triangleleft \sigma_i])}{t \mid [\sigma_s \triangleleft s \triangleleft \sigma_i] \xrightarrow{\bar{t}} [\sigma_s' \triangleleft s' \triangleleft \sigma_i']} & \quad \text{R3'}
\end{align*}
\]

The property below characterizes the relation between the test runs obtained by executing a test case in the synchronous setting and by executing a test case in the queued setting. More precisely, it states that every test run in the synchronous setting is executable in the asynchronous setting which is modeled as queues. This result is not unexpected, because any trace of a system in the synchronous setting can be executed in the queue context as well, see Property 2.8.

Property 2.32. Let \( \langle S, I, U, \rightarrow_s, \bar{s} \rangle \) be an IOTS and let \( \langle T, U \cup \{ \theta \}, I, \rightarrow_t, \bar{t} \rangle \) be a TTS. Consider arbitrary states \( s, s' \in S \) and \( t, t' \in T \) and an arbitrary test run \( \sigma \in L^* \). We have the following properties:

1. \( \bar{t} \parallel \bar{s} \xrightarrow{\sigma} t' \parallel s' \) implies \( \bar{t} \parallel Q(\bar{s}) \xrightarrow{\sigma} t' \parallel Q(s') \)

2. \( \text{Sinit}(\bar{t} \parallel \bar{s}) = \text{Sinit}(\bar{t} \parallel Q(\bar{s})) \).

Internal-choice test cases. As illustrated in the Venn-diagram in the previous section, internal-choice IOTSs represent a new subset of IOLTSs other than IOTSs in which inputs are accepted only at quiescent states. In contrast with the definition of input actions in [Tre96b; Tre96a] where inputs are completely under the control of the environment, in this subclass of IOLTS the initiation of input actions is under the control of the IUT rather than its environment. More precisely, though the environment still decides which input action to be executed, the IUT decides when that input has to be executed. In contrast to IOTSs, an implementation behaving as an IOTS\(^\dagger\) cannot always be stimulated, i.e., a test case may provide an input when the IUT cannot accept it. To mend this, in [Wei09; WW09] a subclass of TTS\((U, I)\) called internal-choice test cases is introduced. Such test cases stimulate an IUT only when quiescence has been observed. Intuitively, this will ensure that the test case is actually executable for implementations that behave as internal-choice transition systems.

Definition 2.33 (Internal-choice test case). A test case \( \langle S, U \cup \{ \theta \}, I, \rightarrow_s, \bar{s} \rangle \) is an internal-choice test case, denoted by TTS\(^\dagger\), iff if for all \( s \in S \), \( x \in I \) and \( \sigma \in L^* \), if \( \sigma x \in \text{traces}(s) \) then \( \sigma = \sigma' \theta \) with \( \sigma' \in L^* \).

We denote the class of internal-choice test cases ranging over inputs \( I \) and outputs \( U \) by TTS\(^\dagger\)(\(U, I\)).
2.3. Input-Output Conformance Testing for IOLTSs

The terminal state(s) **pass** or **fail** of a test case can be used to formalize what it means for an implementation to **pass** or **fail** a test case.

**Definition 2.34 (Verdict).** Let \( T \subseteq \text{TTS}(U, I) \) be a set of test cases for some IOLTS implementation \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) and let \( \bar{t} \in T \) be a test case. We say that state \( s \in S \) **passes** \( \bar{t} \), denoted \( \text{s passes } \bar{t} \) iff there is no \( \sigma \in L^* \) and no state \( s' \in S \), such that \( \bar{t} \semantics{\sigma} \text{ fail} \semantics{s'} \). We also say that state \( s \in S \) **passes** the set of test cases \( T \), denoted \( \text{s passes } T \) iff \( s \) **passes** all test cases in \( T \).

**Remark 2.35.** Note that according to Definition 2.34, the verdict of unsuccessful execution of a test case on the IUT is **pass**.

There are two approaches in model-based testing to present the meaning of conformance between specifications and implementations; defining an explicit formal relation between implementations and specifications, referred to as an implementation relation or defining the conformance notion as a test result of a predefined set of test cases.

Once an implementation relation is defined, the goal of model-based testing is to generate a set of test cases from a given specification to assess the conformance of the IUT with the specification. Various test generation algorithms are presented in the literature for testing different implementation relations. However, the sets of test cases generated by those algorithms have to be **sound** and **exhaustive** with respect to the implementation relation, provided that such a relation is explicitly defined. Soundness means that, for a given specification, executing the test case on an implementation will not lead to a test failure if that implementation conforms to the specification. Exhaustiveness boils down to the possibility of generating a test case that has the potential to detect a non-conforming implementation.

**Definition 2.36 (Soundness and exhaustiveness).** Let \( T \subseteq \text{TTS}(U, I) \) be a set of test cases for IOLTS specification \( \bar{s} \). Then for an implementation relation \( \text{imp} \), we say that

\[
\begin{align*}
T \text{ is sound} &\iff \forall i \cdot i \text{ imp } \bar{s} \implies i \text{ passes } T \\
T \text{ is exhaustive} &\iff \forall i \cdot i \text{ imp } \bar{s} \quad \text{if } i \text{ passes } T
\end{align*}
\]

2.3 Input-Output Conformance Testing for IOLTSs

As mentioned in the previous sections, an implementation relation is a binary relation between implementations and specifications with respect to which the set of implementations is partitioned into the sets of conforming and non-conforming implementations. In this thesis, we mainly consider the input-output conformance relation, for short ioco, introduced by Tretmans in [Tre96c] and an ioco-like implementation relation presented by Weigelhofer in [Wei09].

Following the first approach in defining the conformance notion, Tretmans introduced a conformance relations explicitly between implementations and specification that behave as IOTSs and IOLTSs respectively. To facilitate the presentation, we first introduce the formal definitions below.

**Definition 2.37.** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOLTS. Let \( s \in S \), \( S' \subseteq S \) and let \( \sigma \in L_\bar{s}^* \).
• $s$ after $\sigma = \{s' \in S \mid s \xrightarrow{\sigma} s'\}$, and $S'$ after $\sigma = \bigcup_{s' \in S'} s'$ after $\sigma$.

• $\text{out}(s) = \{x \in L_\delta \setminus I \mid s \xrightarrow{x}\}$, and $\text{out}(S') = \bigcup_{s' \in S'} \text{out}(s')$.

The $\text{ioco}$ conformance relation [Tre96c; Tre08] is then defined as follows.

**Definition 2.38 ($\text{ioco}$).** Let $(Q, I, I, \rightarrow, r, \bar{r})$ be an IOTS representing the IUT, and let IOLTS $(S, I, U, \rightarrow, s, \bar{s})$ be a specification. We say that $\bar{r}$ is input-output conforming with specification $\bar{s}$, denoted by $\bar{r} \text{ioco} \bar{s}$, iff

$$\forall \sigma \in \text{Straces}(\bar{s}): \text{out}(\bar{r} \text{ after } \sigma) \subseteq \text{out}(\bar{s} \text{ after } \sigma)$$

The $\text{ioco}$ relation can be generalized to a family conformance relations by considering different subsets of suspension traces. The family of $\text{ioco}$ conformance relation [Tre08] is formally defined below.

**Definition 2.39 ($\text{ioco}_F$).** Let $(Q, I, I, \rightarrow, r, \bar{r})$ be an IOTS representing a realization of a system, and let IOLTS $(S, I, U, \rightarrow, s, \bar{s})$ be a specification. Let $F \subseteq L_\delta^*$. We say that $\bar{r}$ is input-output conforming to specification $\bar{s}$ over the set of observations $F$, denoted by $\bar{r} \text{ioco}_F \bar{s}$, iff

$$\forall \sigma \in F: \text{out}(\bar{r} \text{ after } \sigma) \subseteq \text{out}(\bar{s} \text{ after } \sigma)$$

The $\text{ioco}_F$ can be realized to a relation by choosing an appropriate $F$. For instance, the standard $\text{ioco}$ relation and $\text{ioconf}$ [Tre96c] are obtained by assigning $F = \text{Straces}(\bar{s})$ and $F = \text{traces}(\bar{s})$ respectively.

**Example 2.40.** Consider the IOLTSs pictured in Figures 2.8. IOLTS $\bar{s}$ is a specification of a vending machine which sells tea. The IOLTS $\bar{r}$ is a formal model of a possible implementation of this vending machine. Upon receiving a coin, the machine $\bar{r}$ chooses nondeterministically between serving tea or refunding the coin. Note that IOLTS $\bar{r}$ is input-enabled, because it accepts input action coin (as the only input action) at every state. The set $\text{Straces}(\bar{s})$ is given by the regular expression $(\delta^+)(\delta^*\text{coin})(\delta^*\text{coin(tea|refund)}\delta^+)$. Clearly, for all $\sigma \in \text{Straces}(\bar{s})$, we have $\text{out}(\bar{r} \text{ after } \sigma) \subseteq \text{out}(\bar{s} \text{ after } \sigma)$. Thus, $\bar{r} \text{ioco} \bar{s}$.

IOLTS $\bar{\ell}$ is a formal model of an implementation of a malfunctioning vending machine. After receiving a coin, it either delivers tea, refunds the coin or does nothing. Similar to IOTS $\bar{r}$, it accepts input action coin at every state. Thus, the IOLTS $\bar{\ell}$ is input enabled. Consider the trace coin after which $\text{out}(\bar{s} \text{ after } \text{coin}) = \{\text{refund, tea, } \delta\}$ while the set of outputs of $\bar{\ell}$ is $\{\text{refund, tea, } \delta\}$. As a result, we find that $\bar{\ell} \text{ioco} \bar{s}$.

It is shown in [Wil06] that for any valid suspension automaton, we can find an IOLTS whose set of conforming implementations is the same as that of the suspension automaton with respect to the $\text{ioco}$ relation.

**Theorem 2.41 (Theorem [Wil06]).** Let $(Q, L \cup \{\delta\}, \rightarrow, \bar{q})$ with $L = I \cup U$ be a valid suspension automaton. Then, there is an IOLTS $\bar{s}$ such that for all implementation $\bar{r} \in \text{IOLTS}(I, U)$, it holds that $\bar{r} \text{ioco} \bar{s}$ iff $\bar{r} \text{ioco} \bar{q}$. 

28
The above proposition and theorem are essential for some of the theorems and proofs presented in this thesis. Suspension automata and IOLTSs may be interchangeably used in this thesis. After introducing the relative quiescence action, we extend the class of suspension automata in Chapter 7 by adding the relative quiescence action to its alphabet.

Following the common practice in model-based testing, Tretmans in [Tre08] presented a test case generation algorithm for the family of the \textit{ioco} conformance relations. He showed in [Tre96c] that the set of test cases generated by that algorithm is sound and exhaustive with respect to a given conformance relation as an instance of the family of \textit{ioco}-conformance relations. Note that in the \textit{ioco} testing theories, it is assumed that test cases are synchronously run on the IUT.

Weiglhofer and Wotawa in [WW09; Wei09] introduced a new \textit{ioco}-like implementation relation, considering implementations that behave as internal-choice input-output transition systems. Checking all set of suspension traces of a given specification on such implementations may be useless, since they cannot always perform inputs. To prevent this from happening, the set of observations in that implementation relation is restricted to a subset of suspension traces in which quiescence is always observed before inputs.

A test case generation algorithm for internal-choice test cases is presented in [WW09; Wei09] for checking the conformance relation, mentioned above, between an implementation and a given specification that behaves as an IOLTS$^\tau$. One must still prove the soundness and exhaustiveness of generated test cases with respect to Weiglhofer’s implementation relation. Moreover, it is claimed, without any formal proof, that the introduced implementation relation obtained from IOTS$^\tau$s and internal-choice test cases have the same testing power as the standard \textit{ioco} relation. We in chapter 4 show that different subclasses of IOLTSs as the behavioral model of the IUT yield different implementation relations.
Chapter 3

Implementing Input-Output Conformance Testing: A Case Study

In this chapter, we report on our experience with applying model-based testing techniques to an operational Electronic Funds Transfer (EFT) switch (also known as Payment Switch, or simply Switch). An EFT switch provides the infrastructure for online financial transactions such as money transfer between bank accounts, electronic payments, balance enquiries, and bill payments. On one side of an EFT switch, there are components that the end-user deals with, such as Automated Teller Machines (ATMs) and Point-of-Sale (POS) terminals, and e-Payment applications. On the other side, there are Core-Banking systems and the inter-bank network connecting EFT switches of different financial institutions. A schematic view of a switch and its environment is given in Figure 3.1.

The EFT system components communicate in the form of transactions consisting of several messages passed through the switch. For example, Figure 3.2 shows two simplified purchase scenarios performed by an EFT switch. Figure 3.2(a) depicts a basic scenario of a purchase transaction in which a purchase transaction originates from a POS terminal,
Chapter 3. Implementing Input-Output Conformance Testing: A Case Study

Figure 3.2: IOLTS $s$ and IOLTS $p$ are the specifications of the EFT switch and a POS terminal respectively in (a) the basic scenario of the purchase transaction, and (b) an exceptional scenario of the purchase transaction. The labeling rule of transactions is as follows: (type of transaction) $(\text{end point})$ $(\text{type of message})$. For example, $\text{pur_pos_rq}$ denotes a request of the purchase transaction receiving from POS terminal, and $\text{rev_core_rs}$ denotes a response of the reverse transaction receiving from the core banking system.

the switch forwards the purchase request to the core banking system (to charge the card holder’s account) and forwards the response back to the POS terminal. In practice, however, possible failures in the components and asynchrony in the communication media may give rise to more complicated transaction flows. For example, Figure 3.2(b) depicts a scenario during which a POS terminal sends a purchase request and it does not receive the response from the switch in time. The POS terminal will then time-out (depicted by a $\tau$-labeled transition) and will send a reversal message to the switch, requesting to cancel the previous transaction. It is also possible that when the time-out occurs, the purchase response is on its way back to the POS terminal. In this case, the POS terminal receives a purchase response after it sends a reversal request (which of course must be responded, too, by the switch). This way, each transaction may comprise a complex combination of different possible interaction scenarios among the components of the EFT system.

In the presence of such complicated transaction flows, a thorough testing of EFT switches is essential, because errors may lead to inconsistencies among different accounts (particularly among accounts at different banks). Such inconsistencies call for a reconciliation process, possibly requiring manual checks, which are very costly for financial institutions.

The correct behavior of a typical EFT system is specified in the ISO 8583 standard [ISO08] at a high level of abstraction. Since the nature of the system is concurrent and distributed, generating test cases manually with a high coverage is practically impossible, as the number of (combinations of) transaction flows is very large. To solve this, we use \textit{model-based testing} as a systematic method to automatically generate test cases from the
Our testing method is mainly based on a formalization of the ISO 8583 standard in terms of input-output labeled transition systems (IOLTSs). Our formal specification captures the behavior of an ISO-compliant EFT switch as well as its environment, i.e., the terminals and the core banking system. We have also performed model-checking on our formal model to make sure that our formalization of the ISO 8583 standard meets the intuitive requirements set forth by the standard as well as by the switch designers. This formalization paves the way to exploit a formal conformance testing method to automatically generate test cases and to perform online conformance testing. We combine ioco testing with functional testing techniques, à la category-partition method, to capture the data-related aspects of switch functionality. Moreover, we interface the test case generator, with our own test-case analysis and execution tool to evaluate, store, and prioritize test cases; the test-cases are executed and their outcomes are also stored by the same tool. Our test selection technique combines the black-box nature of ioco (focusing on model-coverage criteria) with white-box coverage metrics in order to choose an effective test-suite. We developed a prototype tool implementing the above mentioned functionality. Using our tool, we can generate a prioritized test-suite for off-line and regression testing, without any need to explore the formal model any more. Furthermore, during the execution of test cases, our tool also validates various business rules which could not be captured in the formal model.

Structure of the chapter. Section 3.1 provides a background on the switch specification as described in the ISO 8583 standard. Section 3.2 covers our testing approach and describes the way we model the system in terms of IOLTSs. Various aspects of our testing method including test case execution, generation, and prioritization of off-line test suites, as well as checking business rules are also presented in Section 3.2. The test results and code coverage are given in Section 3.3. The merits and demerits of the current approach are discussed in Section 3.4. Finally, we conclude the chapter in Section 3.5.

3.1 EFT Switch Functionality

Typical functionality of an EFT Switch includes performing a purchase, balance enquiry, withdrawal, bill payment, refund, and money transfer. All these functions are composed of a few transaction flows introduced below. Apart from financial functions, there are also features for switch administration, monitoring and auditing that are out of the scope of this study.

As the components of an EFT system are usually provided by different vendors, the ISO 8583 standard [ISO08] is defined to determine the type and the format of the messages exchanged among the components of an EFT system. The standard also defines message and transaction flows at a high level of abstraction. For example, Figure 3.3 shows the flow of a financial transaction as depicted in the standard [ISO08]. According to the standard, the acquirer is defined as “the financial institution (or its agent) which acquires from the card acceptor the data relating to the transaction and initiates that data into an interchange system.” The card issuer is “the financial institution (or its agent) which issues the financial transaction card to the card holder.” According to the described flow,
the acquirer sends a request to the card issuer, followed by zero or more request repeat messages, until it receives a response from the issuer. The data format of the messages (e.g., 1200 - financial request) has been defined elsewhere in the standard [ISO08, Chapter 4]. Note that each typical functionality of an EFT switch, e.g., a purchase or a balance enquiry, is composed of a number of transaction flows, such as the one depicted in Figure 3.3. Apart from the flow depicted in Figure 3.3, there are eleven more transaction flows specified in the standard. We refer to [ISO08, Chapter 5] for a detailed presentation of all transaction flows. In addition to a generic flow for each transaction type, the standard also specifies two other flows. One that specifies possible sequences of transactions relating to a single instance of business at a point of service, and another that defines the reconciliation process between the acquirer and the card issuer. Figure 3.4 shows the former, which can be seen as a high level sequence diagram referencing the individual transaction flows describes before. Each box in the figure is a transaction with its flow specified in the standard, e.g., the financial transaction described before.

### 3.2 Testing the EFT Switch

For this experiment, we used UPPAAL TRON [LMN04; Hes+08; Mik12], an extension of UPPAAL [LPY97; Beh+06; Beh+11] for black-box testing, in order to benefit from several modeling, simulation, verification and test-case generation tools available in its tool-set. UPPAAL is an integrated tool for modeling, validation and verification of real-time systems which is successfully used in case studies varying from communication protocols to multimedia applications, see [Beh+11] and the references therein. A model in UPPAAL is in the form of a network of timed automata extended with data types (bounded integers, arrays, etc). A timed automaton is a finite-state machine (FSM) extended with clock variables, \textit{i.e.}, a set of \textit{locations} which are connected via \textit{edges}, extended with (constraints on and
3.2. Testing the EFT Switch

Figure 3.4: Allowed sequence of business transactions – solid arrows indicate the typical order, while dashed arrows indicate the possible (exceptional) order of transactions [ISO08, Section 5.3.1].

assignments to) clock variables [Beh+06]. An edge in an UPPAAL timed-automata can be annotated by four types of labels: selections, guards, synchronizations and updates. The semantic of a timed automaton is expressed in terms of an (timed) IOLTS [Hes+08].

When taking a transition specified by an edge, an automaton may send or receive a signal in the synchronization part. Synchronization in UPPAAL can be either a handshaking or a broadcast synchronization. Common to general practice, a send signal in UPPAAL is annotated by an exclamation mark and its receive counterpart is annotated by a question mark.

In order to specify concepts such as guards, parameterized synchronization and updates, one can define variables of given finite types in UPPAAL. A variable may be global (visible to all automata), local (visible only locally) or bound (visible only during a transition and assigned a non-deterministic value from a specified range). UPPAAL also provides variable and signal arrays. Signal arrays can be used in order to pass data with signals. One may further define user-defined functions to manipulate and calculate data values in a C-like syntax.

When a transition corresponding to an edge is taken, the update label will be executed and the corresponding variables are thus updated. It is possible to assign an arbitrary value from an integer interval or a scalar set to a bound variable. This is achieved by specifying a select label for the variable.

UPPAAL TRON implements a variant of ioco, called rtioct, that is suited for conformance testing of real-time systems. However, none of the specifications used in this case study use time constraints. As discussed in [ST08], the notion of conformance induced
by rtioco, even when no time constraints are used, differs from that of the ioco relation. The rtioco relation allows a tester to observe an output within the corresponding defined time. In case no time constraint is defined for an output, the tester can wait forever for that output; thus, quiescence is always considered as one of the allowed outputs in this situation in the rtioco relation. However, quiescence is among the allowed observable outputs in the ioco relation only when a quiescent state is reachable; see [ST08] for a formal comparison of the notions. Since, in practice observing quiescence is often implemented by setting a time-out for testers, this theoretical difference does not lead to different test results in practice when time-unbounded specifications are used.

Given that all behaviors of the system under test are not always possible in every environment, UPPAAL TRON uses a separate model for the environment with which the IUT interacts, in order to take into account the environmental constraints in testing. Modeling the environment explicitly, UPPAAL TRON tests only relevant scenarios for the given environment. In this sense, test cases are generated from the composition of the system models with its environment.

### 3.2.1 The Testing Infrastructure

An overview of our test infrastructure is given in Figure 3.5. We implemented this infrastructure in three phases, explained below.

![Figure 3.5: An overview of the test infrastructure](image)

In the first phase, we made an IOLTS model of the EFT switch system and a POS terminal as its environment. The details of this phase are described in Section 3.2.2. Then, we ran the generated test cases on a switch system connected to an operational core banking system (Figure 3.6(a)). We used the timed automata language of UPPAAL as our modeling language and UPPAAL TRON as our test-case generation tool. We developed an adapter to translate and augment abstract interactions of the model to concrete network
messages sent to the switch, on one side, and strip down network messages from EFT to model interactions, on the other side. After running each test-campaign and receiving a verdict from the IOCO testing, we made a manual inspection of the log files. In the initial stages of testing, this resulted in one of the following conclusions:

1. a false negative: the io oc testing reported a failure while the system behavior was as expected; in our experience, such false negatives mostly stem from an incomplete treatment of asynchronous messages in our specification. By making the specification complete, we gradually got fewer and fewer false negatives.

2. a false positive: the io oc testing passed the success verdict while the actual behavior of the system was not as expected (e.g., the current state of the core banking system did not show the expected result of the transaction). This was mainly due to failures in other components of the EFT system, which fell outside our test-plan. We partially solved this problem in our second phase of modeling (see below).

3. a true negative: the io oc testing found a bug. For example, we found a bug which was known to the development team, but was deemed non-reproducible before. Again, the main cause of difficulty in the reproduction of such bugs is the concurrent and asynchronous nature of the system which allows for a huge space of interleaving among different phases of transactions. However, with a precise specification of the interleaving at hand, we could easily reproduce the bug and in debugging, it was traced back to a null pointer dereferencing.

Figure 3.6: Three different possible strategies in testing an EFT switch, see [FT06]: (a) the switch tested while it connects to a real core-banking system with a POS terminal as its environment, (b) the switch tested with a POS terminal and the core-banking system as its environment, (c) the switch tested with a stub of the core-banking system and a POS terminal as its environment.

In the second phase, we first built a model of an ideal POS and the core banking system, acting as the environment for the test-case generator, illustrated in Figure 3.6(b). This step is motivated by that the core banking system is a separate component and usually
operates in a different environment. The core banking system may have its own bugs that should not affect the switch testing process. Hence, using the models for our environment components, we could test a standalone EFT switch. This helped us focus the process on our test plan, eliminate false positives, and isolate the bugs in the debugging procedure. But we faced severe performance problems in this approach when we tried to generate massively parallel executions of transaction flows. After trying several options, the main cause was traced back to the huge state-space of the model resulting from the combination of the core banking system model on one hand and the EFT switch and POS models on the other hand. Hence, we separated the model of the ideal core banking system and implemented it as a separate component, running in parallel, depicted in Figure 3.6(c).

Finally, in the third phase, we built our tools for storing test-cases and their outcomes, prioritizing them and executing off-line test-suites and placed them around the test infrastructure. For test prioritization and test selection, we implemented our heuristics and combined them with the code coverage metrics from Cobertura [Cob14]. This allowed us to reuse the information resulting from an online test campaign in future tests and also to use the generated test-suite for regression testing.

### 3.2.2 Modeling

The behavior of an EFT switch and its environment is specified in terms of a number of transaction flows. Combining all of these flows into a single model (a timed-automaton) would compromise readability and maintainability; it is also very difficult, if not impossible, to check whether the specified automaton is a correct formalization of the flow specified by the ISO standard. Hence, we break the specification into several timed automata, each modeling the behavior of EFT system components in a specific transaction flow (see Figure 3.7).

To show how we model a transaction flow, consider a simplified model of the switch in Reversal transaction, which is used as a cancellation transaction (Figure 3.7(b)). The starting state is $s_1$, where we send a $rev_{\text{ready}}$ signal indicating we are ready to start a Reversal. Then, we wait to receive a signal from a POS terminal to start the transaction. The signal is accepted from the channel $rev_{\text{req}}[j]$. The parameter $j$, determined by the environment, indicates the value of the current transaction ID and is assigned to the variable $\text{curTrx}$ for future use. Specified as a requirement, the response of a reversal request should be immediately sent back to the POS ($rev_{\text{pos rs}}$). It is possible that the response is lost on the way back to the POS. Hence, an unlabeled transition to $s_4$ is also possible. The choice between the two cases is non-deterministic. Making a transition from $s_4$ to $s_5$, the switch sends a reversal request to the core banking system ($rev_{\text{core rq}}$). Note that the two last signals are sent with the same transaction ID (since the switch does not change the transaction ID). However, it is possible to receive a reversal response from the core banking system ($rev_{\text{core rs}}$) with an arbitrary transaction ID. Thus a separate variable is used as the channel parameter ($\text{receiveID}$). In case the received transaction ID is different from the expected value ($\text{curTrx}$), we go back to $s_5$ and wait for the signal with the expected ID. Otherwise, we complete the transaction by sending a $rev_{\text{done}}$ signal. The automaton in Figure 3.7(c) shows the behavior of POS in the same transaction flow.
3.2. Testing the EFT Switch

<table>
<thead>
<tr>
<th></th>
<th>Switch</th>
<th>POS</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance Inquiry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purchase</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reversal</td>
<td>(b)</td>
<td>(c)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.7: (a) Each automaton in the model specifies the behavior of a component in a transaction flow. (b) Simplified models of the behavior of a switch and (c) a POS in the reversal transaction flow.
Chapter 3. Implementing Input-Output Conformance Testing: A Case Study

The POS instance sends a reversal request via the channel $rev_{\text{req}}$ to the switch instance and receives the reversal response via the channel $rev_{\text{pos\_rs}}$.

It is possible to have multiple instances of the same transaction flow executing concurrently. So, we need to have multiple instances of the corresponding automata in our model. This is possible in UPPAAL, since we can declare multiple instances of the same automaton “template”. In fact, the number of declared instances of an automaton determines the maximum number of concurrent instances of the corresponding transaction type. To generate various combinations of transaction flows, we use a coordinator automaton. The coordinator non-deterministically selects the next flow to start and sends it the start signal and repeats this continuously as long as a parallel instance is ready to receive the start signal. For example, when the switch is ready to accept another Reversal request, it sends a $rev_{\text{ready}}$ signal to the coordinator. Then, the coordinator sends a $rev_{\text{start}}$ to the POS automaton to start the Reversal (Figure 3.7).

During development of the model, human mistakes may introduce errors in the model. To discover such errors, we take a model-checking approach to verify the model against correctness properties before the testing process. We first formalized a few intuitive correctness properties based on the ISO standard and the intuition of the designer in the temporal-logic-based verification language (which is a subset of CTL [CES86]) provided by UPPAAL TRON (for some properties, we had to augment the model with observer automata in order to compensate for the limited expressiveness of the logic). For example, the following formula is used to verify that every transaction started must eventually be finished.

$$\text{A}\forall \text{(i : int[0, MAX\_TX])}$
$$\text{TransFlow}[i].\text{start} \rightarrow \text{TransFlow}[i].\text{finish}$$

Subsequently, we used UPPAAL verifier to model check the formalized correctness properties.

Due to the combinatorial explosion of the state space of specifications, the performance of the UPPAAL TRON test case generator was extremely low, when it tried to generate test cases for the whole EFT system. To alleviate the state-space explosion problem, we implemented the abstract model of the core-banking system as a separate Java program and ran it in parallel with UPPAAL TRON and its adapter. With this simple improvement, we were able to increase the performance of the test-case generation by a factor of 10. This way, we could generate test-cases for hundreds of concurrent transaction flows for each instance of UPPAAL TRON. This showed that the performance problem we experienced before was due to the size of the state space of specifications and it is not related to the handling of time in UPPAAL TRON.

### 3.2.3 Interfacing Switch and TRON

UPPAAL TRON continuously interacts with the system under test while exploring the IOLTS model. In other words, on-the-fly test case generation is combined with online testing, so that the next step in the test case generation can be determined by the response from the system under test [MLN03; LMN04]. Hence, to interface UPPAAL TRON with the system under test, an adapter has to be implemented, which in its simplest form, communicates the messages between UPPAAL TRON and the system under test (possibly after converting
them to the right format for each side). We implemented such an adapter which translates the rather plain signals of UPPAAL TRON to (from) the elaborate format of financial messages specified by the ISO 8583 standard. In order to perform the translation to the ISO 8583 messages several details (concerning financial data of a transaction) have to be added to the message, which are selected from representative data stored in our sample database (more explanation about this to follow). Besides the format conversion and the addition/removal of financial data, we developed several other components in the adapter which store and prioritize the test-cases in order to reuse them in regression testing. This way, a prioritized test suite is obtained, which can be run efficiently, without the overhead of exploring the formal model. Finally, there are some types of business rules that are hard to capture in UPPAAL TRON models and hence, are applied and verified by a separate component in the adapter.

3.2.4 Classifying and Covering Data Domains

Common to many reactive systems in the financial domain, the EFT switch exhibits complex reactive behavior while also having a data-dependent nature. An effective test method must address and integrate both of these facets. Theoretically, the test cases generated by ioco cover all behavioral scenarios of interaction between the switch and its environment. Hence, we must set the fields of the messages generated by TRON to different combinations of values. This results in multiple sequences of messages made from the single sequence of messages generated by TRON. Although the UPPAAL modeling language supports some basic data structure, it is still hard to express data constraints on complex message structure and contents. The problems we faced in adding data variables to our models can be enumerated as follows: firstly, data types supported by UPPAAL are much simpler than our needs. To make the problem worse, TRON supports fewer data types than UPPAAL; TRON implements only a subset of data types of UPPAAL. Secondly, though symbolic IOLTSs (a variant of IOLTS extended by data, see [RBJ00; FTW04]) are accepted as input models in some testing theories, e.g., sioco (a variant of ioco extended by data) [FTW06], and a number of testing tools, test cases are still generated from plain IOLTSs. To this end, prior to test case generation, the corresponding semantic IOLTS of each symbolic IOLTS is constructed by assigning all possible values to variables. This approach suffers from a major performance problem along with the state-space explosion problem.

After trying several options, to manage the complexity of the data domain, we have used the classification tree method [GG93] (as an extension of the original category-partition method) to organize the test case generation process. According to the method, we should select an aspect relevant to the test and partition the input domain into disjoint subsets called classes. The resulting classes will be subsequently classified according to some other aspect recursively, resulting in a tree of classifications and classes. This way, we specify representative elements for the content of data elements present in the structure of financial messages.

Moreover, to evaluate the quality of our test cases, we divide the ongoing pattern of interaction into discrete pieces; in the remainder of this chapter, we refer each of these pieces (with some reuse of terminology) as a test case. Hence, a test case is a combination of transaction flows (possibly of different types) with specified values for the data items.
in the messages passed during each transaction. For example, a test case may comprise a purchase transaction succeeded by a reversal. To specify discrete test cases, in addition to the content of the financial messages, we should also specify the type of transaction flow and the size of transactions (repetition of messages and flows). We reuse the same concept of classification-tree to classify and specify representative values for these aspects, as well.

In our prototype implementation, we used the domain and the implementation knowledge of the EFT switch to classify the following set of data domains: transaction flow types, PIN validity, transaction amount, and test-case size.

For each aspect, we select a suitable set of discriminating values by using the domain knowledge. For Transaction type we consider five different classes: Purchase only (P), Balance Inquiry only (B), Purchase and Balance Inquiry (PB), Purchase with Reversal (PR), and Purchase with Reversal and Balance Inquiry (PRB). The PIN validity classification shows whether the transaction is authorized to be executed according to the PIN number input parameter. The domain of Transaction amount is the set of positive integers. The negative and zero cases are also included to test invalid cases. Finally, the Test-Case size parameter is the number of transactions in the test case that determines the size of each transaction. A part of the resulting classification tree is shown in Figure 3.8. Note that the classification tree is not supposed to be a balanced tree and hence, some parameters may not apply to all cases. For example, a Balance Inquiry transaction does not have a transaction amount as an input parameter.

The transaction type parameter is applied to the coordinator automaton (see Section 3.2.2) which affects the sequence of transactions generated by TRON. The other parameter values are set by the adapter. The data selection tree is hard-coded in the adapter.
### 3.3 Test Results

Apart from online testing, which has been very helpful in revealing defects, defining suitable test cases enabled us to measure test coverage for each test case and prioritize the test cases according to our test plan (in this case: full statement coverage of functional components, i.e., components involved in the realization of functionality in the main transaction flows). This prioritized set is used for off-line and regression testing, particularly when running the whole test-infrastructure is not feasible and the testers have to choose some of test cases to get the most coverage. In this work, our test plan is to cover different flows as much as we can, instead of trying to cover all features of the switch.

We have selected the test cases based on our category-partitioning analysis. Due to some obstacles in the implementation, in this work, we have just used positive values for the *Amount* parameter. Other parameters are tested as described in the resulted classification tree (Figure 3.8). We have measured the statement coverage using Cobertura [Cob14]. To reduce measurement errors, each test case has been repeated four times (with the same configuration) and the average coverage is reported in percentages in Table 3.1.

Early analysis showed that there is a considerable amount of common code between the purchase and balance enquiry implementation because of the inherent common logic. This hypothesis is substantiated by measuring the relative coverage of adding a test case of former type to a test case of the latter type (or vice versa); namely, the addition of each type of test case to the other does not significantly increase the statement coverage measure. Hence, combining these two tests (i.e., the PB row in Table 3.1) did not result in any considerable improvement in coverage.

Further analysis shows that a significant amount of code for processing a transaction is devoted to common tasks such as authorization and packet routing. This justifies why there is not much difference between the coverage results of the cases.

Note that the increase in the size of transactions beyond 15 messages did not increase the coverage considerably, since apparently this does not lead to any new behavior in the

<table>
<thead>
<tr>
<th>Trx No.</th>
<th>5</th>
<th>15</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.614</td>
<td>0.622</td>
<td>0.620</td>
</tr>
<tr>
<td>B(Invalid PIN)</td>
<td>0.487</td>
<td>0.487</td>
<td>0.487</td>
</tr>
<tr>
<td>P</td>
<td>0.589</td>
<td>0.589</td>
<td>0.594</td>
</tr>
<tr>
<td>P(Invalid PIN)</td>
<td>0.487</td>
<td>0.487</td>
<td>0.487</td>
</tr>
<tr>
<td>PB</td>
<td>0.596</td>
<td>0.596</td>
<td>0.592</td>
</tr>
<tr>
<td>PB(Invalid PIN)</td>
<td>0.487</td>
<td>0.487</td>
<td>0.487</td>
</tr>
<tr>
<td>PR</td>
<td>0.630</td>
<td>0.671</td>
<td>0.671</td>
</tr>
<tr>
<td>PRB(Invalid PIN)</td>
<td>0.529</td>
<td>0.529</td>
<td>0.529</td>
</tr>
<tr>
<td>PRB</td>
<td>0.712</td>
<td>0.710</td>
<td>0.712</td>
</tr>
<tr>
<td>PRB(Invalid PIN)</td>
<td>0.529</td>
<td>0.529</td>
<td>0.529</td>
</tr>
</tbody>
</table>

Table 3.1: Statement coverage results in percent
EFT switch and the same logic is executed repeatedly. However, in our experience, having a large number of parallel transaction instances does increase the chance of catching errors caused by concurrency issues or (thread-pool) overflow problems.

Another point is that test cases with exceptions have lower coverage among other combinations, yet they are deemed very important by domain experts. This is true because the switch drops unauthorized messages in early stages, so a big part of the code will never run. This is justified by developers’ insight that the code for handling exceptional cases has little overlap with the code for the normal transaction flows. Hence, despite their individual low coverage, these test-cases should appear with high priority in the final priority list.

3.4 Discussion

Our system under test is inherently a mixture of reactive and functional behavior: it implements a high-level protocol for exchanging messages for a financial transaction, while its detailed implementation is very much dependent on the functional and data-related aspects. This mixture, if not structured properly, makes the generated models overly cluttered and complicated. Unfortunately, the \texttt{ioco} testing theory (and its different variants) and consequently, most of the existing \texttt{ioco}-based tools, including \texttt{UPPAAL TRON} and \texttt{TorX} \cite{TB03} do not provide proper facilities for orthogonalizing, structuring and relating reactive and functional behavior, and do not support asynchronous and message-based communications. Hence, we conducted research on model-based conformance testing to improve \texttt{ioco} testing theory in order to solve the following issues:

1. Asynchronous message passing: We experimented with different additions to our model in order to cater for the asynchronous nature of communication in our domain. We first tried adding input/output queues, which immediately led to drastic performance drawbacks. Then, we experimented with abstracting from the asynchronous delays in our protocols, which did lead to better performance. However, such an abstraction resulted in fictitious sequences of messages that are not expected by the IUT. To overcome this, we had to add several guards to guarantee that the model will only be triggered with appropriate signals. This last modification has led to a complicated specification. An inherent support for asynchronous message passing may be considered as an option, along the lines of the initial proposal in \cite{WW09}. Hence, this observation motivated our study in Chapter 4, where we undertake a comprehensive study of the notion of conformance introduced in \cite{WW09}. We subsequently study in Chapter 6 the necessary and sufficient conditions under which \texttt{ioco} test cases are still meaningful when a tester has to interact with the IUT asynchronously.

2. Test selection: Although the types of financial transactions defined in ISO 8583 are rather limited, due to the concurrent nature of the system, the behavioral model of the system from which test cases are generated is very large and complicated. To guarantee the correctness of the IUT (according to a conformance relation, here the \texttt{ioco} relation), we had to execute the exhaustive set of \texttt{ioco} test cases. This set is infinite in many large industrial systems, including the EFT switch. Thus, testing
all possible test cases is impractical. Having good test selection criteria therefore is essential. In Chapter 5, we study the complexity of the \textbf{io:co} relation to find the theoretical limitations of this implementation relation. We believe that in the light of our results in that chapter, we can identify assumptions on implementations that allow to put a constraint on the length of test cases. In this way, we would be able to select a finite set of \textbf{io:co} test cases with the same power of recognition with that of the (possibly infinite) exhaustive set of test cases.

3. Performance issues: Due to the very complex and mixed nature of the system, we soon reached the boundaries of possibilities with \texttt{UPPAAL TRON}. To overcome this problem we had to distribute our test case generation among a number of parallel instances of \texttt{UPPAAL TRON}. A challenge imposed by this solution was how to pass the received messages to the right instance of \texttt{UPPAAL TRON}. This problem was intensified by the lack of appropriate support of data-type-handling. To solve the latter problem, we annotated the messages in the underlying model of each \texttt{UPPAAL TRON} instance with a unique identifier which can be recognized and distinguished by our adapter. However, this solution cannot be considered as a proper one: firstly, some parts of test execution are hard-coded in the adapter (and also in our models). But the adapter is supposed to be a light-weight program that only connect the tester with the IUT by translating abstract actions of models into concrete messages of the real system, and vice versa. Secondly, models in this solution tend to get larger and more complex, because irrelevant data to the behavior of the IUT is added to them. Thirdly, this solution is not scalable per se. Considering that so far we have experimented with few types of transactions, this problem is more critical when our specification becomes larger by modeling the behaviors of the remaining transactions. By breaking a large and complex specification into smaller and simpler models in such a way we may still be able to reason about the correctness of the whole system when those smaller specifications are used can be considered as an option. This motivates our study in Chapter 7 where we investigate under which conditions, a specification of a sub-system can be derived from a given specification of the whole system with respect to other components of the system assumed to be well-known (and well tested).

3.5 Closing Remarks

In this chapter, we reported on our experience in applying model-based testing methods on a high-risk financial system, called Electronic Fund Transfer (EFT) switch. Our goal in this work is to test the conformance of the EFT switch to the standard flows described by the ISO 8583 standard. We used an existing testing tool, called \texttt{UPPAAL TRON}, to generate test cases. In addition to the adapter used to connect the tester with the IUT, we developed several components to augment the test cases with concrete test data and to store and prioritize test cases. Also, to enhance the performance and to prevent state-space explosion in our testing infrastructure, we implemented the formal model of some components in the environment as a separate Java component running in parallel with our test infrastructure.
Although in this work, we covered only a small number of financial transaction flows specified in ISO 8583 standard, our test results in terms of the code coverage and the number of bugs found are encouraging. However, this experience shows that there are still some major obstacles in applying model-based testing in large-scale industrial software systems. To overcome those obstacles some of which we encountered during our experience in testing the EFT switch, we need to improve some theoretical aspects of existing conformance testing theories, specifically the \texttt{ioco} relation. Hence, we set the main objective of this thesis to the development of the required theories along the lines presented in Section 3.4.
Chapter 4

Implementation Relations

Many testing theories, including the ioco testing theory, assume that the communication between tester and IUT is synchronous. Asynchronous testing concerns a situation in which the IUT interacts with its environment through asynchronous communication channels. Uncontrollable delays occurring in asynchronous communication cause distortion in observations of the behavior of the IUT. Therefore, the verdict delivered in asynchronous testing may be different from the one reached in synchronous testing. Hence, the set of test cases used in synchronous settings may not be sound to use in asynchronous communication.

An intuitive solution for adjusting synchronous testing theories to the asynchronous setting is to take into account the test execution context [Vil+12, Chapter 8]; in this case the queue context [Ver+93], see Section 2.1.1. The composition of a given specification with queue models is considered as the specification of the IUT communicating via queues. Test cases, consequently, are generated from that new model. Besides the infamous problem of state space explosion, the aforementioned solution affects the power of testing. For instance, a correct implementation in synchronous testing may be rejected in asynchronous testing, see Section 3.4.

To sidestep the aforementioned problems in the ioco framework, in [Wei09; WW09] a subclass of IOLTSs is introduced, namely internal-choice labeled transition systems (IOLTS[^1]). An IOLTS[^1] model represents the behavior of a system in which the input actions can be accepted only when the system is stable, i.e., at quiescent states. In [Wei09; WW09], a class of test cases is also presented to capture such behaviors. These test cases are called internal-choice test cases. Such test cases, therefore, do not stimulate the IUT unless quiescence is observed.

The relation between the standard set of ioco test cases and the set of internal-choice test cases has not been studied. Furthermore, the intentional definition of conformance notion underlying internal-choice test cases has not been presented in [Wei09; WW09]. Knowing that a different set of test cases checks various implementation relations, we pose a fundamental question about the conformance notion obtained from internal-choice test cases. To this end, in this chapter we present a novel intensional representation of the conformance testing relations that result from internal-choice IOLTS[^1]s. Using this representation, we conduct a comprehensive study comparing and contrasting the testing
power of those implementation relations with the \textit{ioco} family of conformance relations. The study in this chapter is carried out in the synchronous setting as the standard setting. The relation between those implementation relations in the asynchronous setting is investigated later in Chapter 6.

**Structure of the chapter.** We present in Section 4.1 a unifying intensional definition of input-output conformance testing, from which the different conformance relations presented in [Wei09; WW09] and [Tre08] can be obtained as special cases. Then, in Section 4.2 we investigate the impact of using different models on each conformance relation. In the same section, we define a notion of testing power and using that compare several notions of conformance relation obtained from different hypotheses assumed in [Wei09; WW09] and [Tre08]. In Section 4.3, we compare the testing power of different instances of implementation relations obtained from various sets of observations. In Section 4.4, we present corresponding extensional notions of conformance testing using test cases and show that they are indeed sound and exhaustive with respect to their intensional counterparts. We finally conclude the chapter with some closing remarks.

### 4.1 Input-Output Conformance Relations with Quiescence

In formal testing, an implementation is said to be correct when its executions are among those prescribed by its formal specification. As we explained in Chapter 2, we assume that implementations (and their behaviors) can be modeled by an IOTS (or IOTS\(^\dag\)). This assumption allows one to formalize the notion of conformance. Tretmans formalized in [Tre08] a family of conformance relations by parameterizing a single behavioral relation with a set of decorated traces. We generalize this family of conformance relations by parameterizing it with the behavioral models it assumes as implementations and specifications. In this chapter we consider only non-divergent implementations. This assumption does not impose any restriction on our family of conformance relation, as it is already presumed by Tretmans in \textit{ioco} family of conformance relations.

**Definition 4.1 (ioco\(_F^{a,b}\)).** Let \(a, b \in \{\cap, _\}\) and let \(\tilde{r}\) be an IOTS\(^a\), \(\tilde{s}\) an IOLTS\(^b\), and \(F \subseteq L^*_\delta\). We say that implementation \(\tilde{r}\) is input-output conforming to specification \(\tilde{s}\), denoted by \(\tilde{r} \text{ ioco}\(_F^{a,b}\) \(\tilde{s}\), iff

\[
\forall \sigma \in F \bullet \text{out}(\tilde{r} \text{ after } \sigma) \subseteq \text{out}(\tilde{s} \text{ after } \sigma)
\]

**Remark 4.2.** Note that \(_\) defines the blank space, that is, for \(a = _\) we have IOTS\(^a = IOTS\).

If we assume that our implementations can be modeled as IOTSs, the family of conformance relations \textit{ioco}\(_F^{a,b}\) reduces to the family of conformance relations \textit{ioco}\(_F\), studied by Tretmans [Tre08]. By setting \(F\) to Straces(\(\tilde{s}\)) for a given specification \(\tilde{s}\), the conformance relation \textit{ioco} [Tre08] is obtained.

In the remainder of this section, we compare and contrast several instances of the \textit{ioco}\(_F^{a,b}\) testing theory. First, we verify in Subsection 4.2 that restricting the class of specifications in the \textit{ioco}\(_F^{a,b}\) relation affects the discriminating power of the implementation relation, henceforth referred to as the testing power. Then, in Subsection 4.3, we study...
the impact of different observations on the power of testing with respect to the family of $\text{ioco}^{a,b}$. 

## 4.2 The Effect of Models on the Power of Testing

We start by formally defining what it means for two classes of specifications to have equal testing power.

**Definition 4.3.** Let $\text{MOD}_i$ be a class of implementations and let $\text{MOD}$ be a class of specifications. Let $\text{MOD}_s$ and $\text{MOD}'_s$ be two subsets of the class of specifications $\text{MOD}$. Then $\text{MOD}_s$ and $\text{MOD}'_s$ have the same testing power with respect to a given implementation relation $\text{impl} \subseteq \text{MOD}_i \times \text{MOD}$, iff

$$
\forall s \in \text{MOD}_s : \exists s' \in \text{MOD}'_s : \forall i \in \text{MOD}_i : i \text{impl} s \iff i \text{impl} s'
$$

and

$$
\forall s \in \text{MOD}'_s : \exists s' \in \text{MOD}_s : \forall i \in \text{MOD}_i : i \text{impl} s \iff i \text{impl} s'
$$

**Remark 4.4.** Note that if $\text{MOD}'_s \subseteq \text{MOD}_s$, then one of the above implications, i.e., $\forall s \in \text{MOD}'_s : \exists s' \in \text{MOD}_s : \forall i \in \text{MOD}_i : i \text{impl} s \iff i \text{impl} s'$ is trivially satisfied.

Informally, given a class of specifications $\text{MOD}$, two subclasses $\text{MOD}_s$ and $\text{MOD}'_s$ have equivalent testing power when for every specification from $\text{MOD}_s$, we can find an alternative specification from $\text{MOD}'_s$ that identifies exactly the same set of correct and the same set of incorrect implementations, and vice versa. Note that we do not require such an alternative specification to be obtained constructively.

In this section, we compare the testing power of IOLTS$^{\uparrow}$ with that of IOLTSs for different classes of implementations when suspension traces are tested. Note that the class IOLTS$^{\uparrow}$ is a subset of the class IOLTS. Hence, to compare the testing power of these two classes of specification, we need to check if for every IOLTS specification, we can find an IOLTS$^{\uparrow}$ specification with the same testing power. Table 4.1 summarizes our results in this sections. In each row the testing power of IOLTS$^{\uparrow}$ and IOLTS are compared in testing implementations that behave as the mentioned subclass of IOLTS.

<table>
<thead>
<tr>
<th>implementation model</th>
<th>implementation relation</th>
<th>Testing power of IOLTS vs. IOLTS$^{\uparrow}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOLTS</td>
<td>$\text{ioco}^{\text{Straces}(s)}$</td>
<td>Different (section 4.2.1, page 50 )</td>
</tr>
<tr>
<td>IOLTS$^{\uparrow}$</td>
<td>$\text{ioco}^{\text{Straces}(s)}$</td>
<td>Different (section 4.2.2, page 51 )</td>
</tr>
<tr>
<td>IOLTS $\cap$ IOLTS$^{\uparrow}$</td>
<td>$\text{ioco}^{\text{Straces}(s)}$</td>
<td>Same (section 4.2.3, page 53 )</td>
</tr>
</tbody>
</table>
4.2.1 IOLTS vs. IOLTS\(\supseteq\) in Testing Input-Enabled Implementations

Considering different subclasses of IOLTS as behavioral models results in different implementation relations as instances of \(\text{ioco}_{\mathcal{F}}^{a,b}\). The theorem below states that restricting specifications from IOLTS to IOLTS\(\supseteq\) influences the testing power with respect to the implementation relation \(\text{ioco}_{\text{Straces}(s)^{\omega}}\), i.e., the standard \(\text{ioco}\) relation, in which an implementations is assumed to behave as an IOTS.

**Theorem 4.5.** The testing power of IOLTS\(\supseteq\) is not equal to the testing power of IOLTS with respect to implementation relation \(\text{ioco}_{\text{Straces}(s)^{\omega}}\).

**Proof.** We must show that the following statement does not hold:

\[
\forall s \in \text{IOLTS}(\{a\}, \{x, y\}) : \exists s' \in \text{IOLTS}(\{a\}, \{x, y\}) : \forall i \in \text{IOTS}(\{a\}, \{x, y\}) : i \text{ ioco}_{\text{Straces}(s)^{\omega}} s \iff i \text{ ioco}_{\text{Straces}(s)^{\omega}} s'
\]

To this end, we show that there is a set of implementations on which the IOLTS specification’s verdict will always differ from any candidate alternative IOLTS\(\supseteq\) specification.

Figure 4.1: An input-output labeled transition system specification and three implementations over the sets of inputs \(\{a\}\) and outputs \(\{x, y\}\) that together show that conformance testing using internal-choice input-output labeled transition system specifications does not have the same testing power as conformance testing using input-output labeled transition systems (Theorem 4.5).

Consider the specification \(s \in \text{IOLTS}(\{a\}, \{x, y\})\), depicted in Figure 4.1(a). Observe that \(\text{Straces}(s) = \{\varepsilon\} \cup x\delta^{*} \cup ax\delta^{*}\). Next, consider the three implementations \(i_1, i_2\) and \(i_3\), also depicted in Figure 4.1. We have:

- \(i_1 \text{ ioco } s\), as for all \(\sigma \in \text{Straces}(s)\), \(\text{out}(i_1 \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)\).
- \(i_2 \text{ iqco } s\), as we have \(\text{out}(i_2 \text{ after } a) = \{y\}\), whereas \(\text{out}(s \text{ after } a) = \{x\}\).
- \(i_3 \text{ iqco } s\), as we have \(\text{out}(i_3 \text{ after } \varepsilon) = \{x, \delta\}\), whereas \(\text{out}(s \text{ after } a) = \{x\}\).

We show that no IOLTS\(\supseteq\) specification leads to the same partitioning on the set of implementations \(\{i_1, i_2, i_3\}\), and, therefore, also not on the entire set of implementations IOTS. We first show that any IOLTS\(\supseteq\) specification \(s'\) that satisfies \(i_1 \text{ ioco } s'\) must necessarily also satisfy either \(i_2 \text{ ioco } s'\) or \(i_3 \text{ ioco } s'\). More formally, we show that:

\[
\forall s' \in \text{IOLTS}(\{a\}, \{x, y\}) : i_1 \text{ ioco } s' \Rightarrow (i_2 \text{ ioco } s') \text{ or } (i_3 \text{ ioco } s')
\]
Let $s'$ be an arbitrary IOLTS$^\exists$ specification such that $i_1 \mathbf{ioco}s'$. Now, assume that $i_2 \mathbf{ioco}s'$. Towards a contradiction, assume that $i_3 \mathbf{ioco}s'$. We then have $z \in \text{out}(i_3 \text{ after } \sigma)$ and $z \notin \text{out}(s' \text{ after } \sigma)$ for some $z$ and some $\sigma \in \mathbf{Straces}(s')$. Observe that for all $\sigma' \in \mathbf{Straces}(s') \setminus \mathbf{Straces}(i_3)$, we have out$((i_3 \text{ after } \sigma') = \emptyset \subseteq \text{out}(s' \text{ after } \sigma')$, so, necessarily, $\sigma \in \mathbf{Straces}(s') \cap \mathbf{Straces}(i_3)$. We have

$$\mathbf{Straces}(i_3) = \{\varepsilon\} \cup \delta^+ \cup \delta^+ a^+ \cup \delta^+ a^+ x \{\delta, a\}^* \cup x \{\delta, a\}^*$$

We next analyze each of these possibilities.

- Case $\sigma = \varepsilon$. Since $i_1 \mathbf{ioco}s'$, we have $x \in \text{out}(s' \text{ after } \varepsilon)$. As out$((i_3 \text{ after } \varepsilon) = \{\delta, x\}$ and $x \in \text{out}(s' \text{ after } \varepsilon)$, we have $\delta \notin \text{out}(s' \text{ after } \varepsilon)$. But then $a \notin \text{Fin}(s')$, since in $s'$, inputs are only allowed in quiescent states. This means that $s'$ cannot distinguish between $i_1$ and $i_2$, contradicting $i_1 \mathbf{ioco}s'$ and $i_2 \mathbf{ioco}s'$. So $\sigma \notin \varepsilon$.

- Case $\sigma \in \delta^+$. Since after observing quiescence, we are necessarily in a quiescent state, we find that out$((i_3 \text{ after } \sigma) = \{\delta\} = \text{out}(s' \text{ after } \sigma)$. So $\sigma \notin \delta^+$.

- Case $\sigma \in \delta^+ a^+$. Observe that since $s'$ is an IOLTS$^\exists$, we have out$((s' \text{ after } \rho \delta a' \rho') = \text{out}(s' \text{ after } \rho a' \rho')$ for all inputs $a'$. This means that we have out$((s' \text{ after } \sigma) = \text{out}(s' \text{ after } \sigma')$, where $\sigma' \in a^+$ is obtained from $\sigma$ by removing all observations of $\delta$. Since out$((i_3 \text{ after } \sigma) = \{x\}$, we must have $x \notin \text{out}(s' \text{ after } \sigma)$. Since out$((s' \text{ after } \sigma) = \text{out}(s' \text{ after } \sigma')$, we find that $x \notin \text{out}(s' \text{ after } \sigma')$. But that contradicts $i_1 \mathbf{ioco}s'$. So $\sigma \notin \delta^+ a^+$.

- Case $\sigma \in \delta^+ a^+ x \{\delta, a\}^*$. We have out$((i_3 \text{ after } \sigma) = \{\delta\}$, so, necessarily, $\delta \notin \text{out}(s' \text{ after } \sigma)$. Again, since $s'$ is an IOLTS$^\exists$, we obtain that out$((s' \text{ after } \sigma) = \text{out}(s' \text{ after } \sigma')$, where $\sigma' \in a^+xa^+$ is obtained from $\sigma$ by removing all observations of $\delta$. That means that $\delta \notin \text{out}(s' \text{ after } \sigma')$, which contradicts $i_1 \mathbf{ioco}s'$. So $\sigma \notin \delta^+ a^+ x \{\delta, a\}^*$.

- Case $\sigma \in x \{\delta, a\}^*$. Since out$((i_3 \text{ after } \sigma) = \{\delta\}$, we must have $\delta \notin \text{out}(s' \text{ after } \sigma)$. Following the same reasoning as in the previous cases, we find that this contradicts $i_1 \mathbf{ioco}s'$. So $\sigma \notin x \{\delta, a\}^*$.

We find that $i_3 \mathbf{ioco}s'$, because none of the possible traces $\sigma \in \mathbf{Straces}(i_3) \cap \mathbf{Straces}(s')$ can lead to out$((i_3 \text{ after } \sigma) \notin \text{out}(s' \text{ after } \sigma)$.

Summarizing, this means that there is no IOLTS$^\exists$ specification $s'$ that has the same testing power as the IOLTS specification $s$, proving Theorem 4.5. \qed

### 4.2.2 IOLTS vs. IOLTS$^\exists$ in Testing Internal-Choice Implementations

Changing the behavioral models of implementations, we next compare and contrast the testing power of IOLTS$^\exists$ specifications with IOLTSs with respect to implementation relation $\mathbf{ioco}^\exists_{\mathbf{Straces}(s)}$. Similar to Theorem 4.5, the theorem below states that restricting the class of specifications from IOLTSs to IOLTS$^\exists$s has an impact on the testing power over IOLTS$^\exists$ implementations.

**Theorem 4.6.** The testing power of IOLTS$^\exists$ does not equal the testing power of IOLTS with respect to implementation relation $\mathbf{ioco}^\exists_{\mathbf{Straces}(s)}$. 

51
Proof. Similar to the proof of Theorem 4.5, we have to show the following statement does not hold.
\[
\forall s \in \text{IOLTS}(I, U) : \exists s' \in \text{IOLTS}^\cap(I, U) :
\forall i \in \text{IOTS}^\cap(I, U) : i \text{io} \text{co}^\cap_{\text{Straces}(s)} s \iff i \text{io} \text{co}^\cap_{\text{Straces}(s')} s'
\]

Therefore, it is sufficient to find an IOLTS specification for which the set of correct IOTS implementations is not the same as the set of conforming implementations of any IOLTS specification. More formally, we need to show that for some sets of inputs and outputs,
\[
\exists s \in \text{IOLTS}(I, U) : \forall s' \in \text{IOLTS}^\cap(I, U) :
\exists i \in \text{IOTS}^\cap(I, U) : i \text{io} \text{co}^\cap_{\text{Straces}(s)} s \text{ but } i \text{io} \text{co}^\cap_{\text{Straces}(s')} s'
\]

Consider the specification \( s \in \text{IOLTS}(\{a\}, \{x\}) \), pictured in Figure 4.2. Observe that \( \text{Straces}(s) = \delta^* \cup \delta^* a \delta^* \cup \delta^* a a x \delta^* \cup a \delta^* a \delta^* \). Next, we show that the specification \( s \) does not accept any IOTS \( \cap \) implementation over the sets of input \( \{a\} \) and outputs \( \{x\} \). We partition the class of IOTS \( \cap \) models over the sets of inputs \( \{a\} \) and outputs \( \{x\} \) into four categories,

- \( I_1 = \{ i \in \text{IOTS}^\cap(\{a\}, \{x\}) | \text{Straces}(i) \subseteq L_0^* \land x \in \text{Straces}(i) \} \)
- \( I_2 = \{ i \in \text{IOTS}^\cap(\{a\}, \{x\}) | \{x\} \cap \text{Straces}(i) = \emptyset \land ax \in \text{Straces}(i) \} \)
- \( I_3 = \{ i \in \text{IOTS}^\cap(\{a\}, \{x\}) | \{x, ax\} \cap \text{Straces}(i) = \emptyset \land aax \in \text{Straces}(i) \} \)
- \( I_4 = \{ i \in \text{IOTS}^\cap(\{a\}, \{x\}) | \{x, ax, aax\} \cap \text{Straces}(i) = \emptyset \land aa \delta \in \text{Straces}(i) \} \)

Note that the union of \( I_1, I_2, I_3 \) and \( I_4 \) includes all possible internal-choice input-enabled implementations over \( \{a\} \) and \( \{x\} \) as the sets of inputs and outputs respectively, i.e., \( \bigcup_{0 < j < 5} I_j = \text{IOTS}^\cap(\{a\}, \{x\}) \). We observe that

- \( i \text{io} \text{co} s \) for all \( i \in I_1 \), as we have \( \text{out}(i \text{ after } \epsilon) = \{x\} \), whereas \( \text{out}(s \text{ after } \epsilon) = \{\delta\} \).
- \( i \text{io} \text{co} s \) for all \( i \in I_2 \), as we have \( \text{out}(i \text{ after } a) = \{x\} \), whereas \( \text{out}(s \text{ after } a) = \{\delta\} \).
- \( i \text{io} \text{co} s \) for all \( i \in I_3 \), because \( \text{out}(i \text{ after } a \delta a) = \{x\} \), whereas \( \text{out}(s \text{ after } a \delta a) = \{\delta\} \).
- \( i \text{io} \text{co} s \) for all \( i \in I_4 \), because \( \text{out}(i \text{ after } \delta a a) = \{\delta\} \), whereas \( \text{out}(s \text{ after } \delta a a) = \{x\} \).

Figure 4.2: The input-output labeled transition system specification \( s \) over the sets of inputs \( \{a\} \) and outputs \( \{x\} \)

52
Hence, the set of correct implementations of specification \(s\) on \(\text{IOTS}^{\pi}(\{a\}, \{x\})\) is empty. We next show that no \(\text{IOLTS}^{\pi}\) specification leads to the same partitioning on the entire set of \(\text{IOTS}^{\pi}\) implementations. To this end, we show that any \(\text{IOLTS}^{\pi}\) specification \(s'\) necessarily accepts at least one of the implementations of \(\text{IOTS}^{\pi}(\{a\}, \{x\})\). More formally, we always have:

\[
\forall s' \in \text{IOLTS}^{\pi}(I, U), \exists i \in \text{IOLTS}^{\pi}(I, U) \cdot (i \circ \text{Straces}(s') s')
\]

Assume that, we have \(\text{IOLTS}^{\pi} s'\) of the form \(\langle S, I, U, \rightarrow, s' \rangle\) with \(I = \{a\}\) and \(U = \{x\}\). We then construct the \(\text{IOLTS}^{\pi} \langle R, I, U, \rightarrow, R \rangle\) from \(\text{IOLTS}^{\pi} s'\) as follows:

- \(R = S\) is the set of states,
- \(r = s'\) is the initial state, and
- \(\rightarrow_i = \rightarrow_i \cup \{(r', a, r') \mid \exists \sigma \in L^\pi \cdot s' \xrightarrow{\sigma, \delta} r' \land a \notin \text{init}(s' \text{ after } \sigma \delta)\}\) is the transition relation.

Note that the transition relation \(\rightarrow_i\) is the result of extending the transition relation \(\rightarrow_q\) by adding a self-loop for each missing input action at every single quiescent state. We observe that \(\text{out}(r \text{ after } \sigma) = \text{out}(s' \text{ after } \sigma)\) for every sequence \(\sigma \in \text{Straces}(s');\) thus, \(r \circ \text{Straces}(s') s'.\) Therefore, we cannot find any \(\text{IOLTS}^{\pi} s'\) whose set of correct \(\text{IOTS}^{\pi}\) implementations is empty. More formally, we have that:

\[
\forall s' \in \text{IOLTS}^{\pi}(I, U), \exists i \in \text{IOLTS}^{\pi}(I, U) \cdot (i \circ \text{Straces}(s') s') \text{ and } (i \circ \text{Straces}(s') s')
\]

\(\square\)

### 4.2.3 IOLTS vs. IOLTS\(^\pi\) in Testing Input-Enabled Internal-Choice Implementations

So far, we showed that restricting specifications to the subclass of internal-choice models affects the testing power on both classes of implementations. More precisely, the class of IOLTS models, in general, has a finer discrimination power in comparison to the subclass of IOLTS\(^\pi\) models.

In the remainder of this section, we show that the testing power of IOLTS\(^\pi\) is the same as that of IOLTSs for any implementation \(i\) that behaves as an input-enabled internal-choice transition system, i.e., \(i \in \text{IOTS}(I, U) \cap \text{IOLTS}^{\pi}(I, U)\). To this end, we present a constructive solution to build a testing-equivalent internal-choice specification \(\hat{q}\) for a given IOLTS specification \(\hat{s}\) with respect to the \(\text{Ico}\) implementation relation. The IOLTS\(^\pi\) specification \(\hat{q}\) is constructed in several steps.

Before formally defining the IOLTS\(^\pi\) \(\hat{q}\) for the given IOLTS specification \(\hat{s}\), we first explain the intuition behind its construction. We observe that an implementation \(i\) which behaves as an input-enabled internal-choice IOTS is always able to reach a quiescent state after any execution, i.e., \(\forall i \in \text{IOTS}(I, U) \cap \text{IOLTS}^{\pi}(I, U), \forall \sigma \in \text{Straces}(i) \cdot \delta \in \text{out}(i \text{ after } \sigma)\). Therefore, implementation \(i\) is rejected whenever \(\delta\) is not among outputs specified in a given IOLTS specification \(\hat{s}\) after some execution. Thus, testing traces
of specification \( \bar{s} \) that contain an input action before which observing quiescence is not possible is redundant when the implementation \( i \) belongs to \( \text{IOTS}(I, U) \cap \text{IOTS}^\cap(I, U) \). This observation is the intuition behind our solution in constructing the \( \text{IOTS}^\cap S \) from the IOLTS \( \bar{s} \). Hence, an input transition is added to \( \text{IOTS}^\cap q \) at a quiescent state if and only if observing quiescence is permissible in IOLTS \( \bar{s} \) before executing the input action. Therefore, two different states which are reachable from the initial state \( \bar{s} \) with the same sequence of input and output actions but with possibly different moments of observing quiescence are not distinguishable from each other in \( \text{IOTS}^\cap q \); such states are accumulated in one state of \( \text{IOTS}^\cap q \). An output action including quiescence is added to a state in \( \text{IOTS}^\cap q \) if and only if all its accumulating states produce that output in IOLTS \( \bar{s} \); an output is added to \( \text{IOTS}^\cap q \) if it is allowed after all traces with the same input and output actions, regardless of the moment of observing quiescence in IOLTS \( \bar{s} \). To distinguish between a real quiescent state in \( \text{IOTS}^\cap q \) with a state whose accumulated states do not agree on any output actions, we introduce a fresh output action \( \sigma \) in the first step, i.e., \( \sigma \notin U \). We add a \( \sigma \)-labeled self-loop to states that cannot produce any output, including quiescence. Notice that no input action can be taken at those states, because they do not allow any output, including quiescence. We then introduce some transformation functions on \( \text{IOTS}^\cap q \) to remove those \( \sigma \)-labeled transitions while the testing power of \( \text{IOTS}^\cap q \) is preserved.

Adding the fresh action \( \sigma \) to the set of outputs, in the following proposition, we build the testing-equivalent \( \text{IOTS}^\cap q \) from the IOLTS \( \bar{s} \) with respect to the \text{ico} relation.

**Proposition 4.7.** Let \( L = \text{I} \cup U \), and let \( \langle S, L, \rightarrow_\text{s}, \bar{s} \rangle \) be a suspension automaton of an IOLTS. Let \( \langle Q, I, U \cup \{\sigma\}, \rightarrow_\text{q}, \bar{q} \rangle \) be an IOLTS where \( Q = \mathbb{P}(S) \cup \{q_\delta | q \in \mathbb{P}(S)\}, \bar{q} = \{\bar{s}\}, \sigma \) is a fresh output action, i.e., \( \sigma \notin U \cup \{\delta\} \), and \( \rightarrow \) is defined by the following inference rules:

\[
\begin{align*}
q &\in \mathbb{P}(S) \quad \forall s \in q \cdot s \xrightarrow{a} s' \quad a \in L, \quad R_1 \\
q &\in \mathbb{P}(S) \quad \forall x \in U \cup \{\delta\}, \exists s \in q \cdot s \xrightarrow{x} s' \quad R_2 \\
q &\in \mathbb{P}(S) \quad \exists s \in q \cdot s \xrightarrow{\delta}, q_\delta \quad R_3
\end{align*}
\]

then,

1. \( \text{IOLTS} \bar{q} \) is an \( \text{IOTS}^\cap \)

2. \( \forall \sigma \in \text{Straces}(\bar{q}), \) there exists \( q \in \mathbb{P}(S) \) such that \( (\bar{q} \text{ after } \sigma) \subseteq \{q, q_\delta\} \).

3. \( \forall \sigma \in \text{Straces}(\bar{q}) \cap (L_\delta^* L), \) there exists \( q \in \mathbb{P}(S) \) such that \( q \in (\bar{q} \text{ after } \sigma) \).

**Proof.** Each item is discussed separately.

1. Since no output action is enabled at state \( q_\delta \) for all \( q \in \mathbb{P}(S) \), \( q_\delta \) is quiescent. Also note that \( q_\delta \)'s are the only states that permit an input action (inference Rule \( R_3 \)). Therefore, \( \text{IOLTS} \bar{q} \) is an internal-choice IOLTS.

2. It is straightforward by induction on the length of \( \sigma \in \text{Straces}(\bar{q}) \).

3. We know from the second item that \( (\bar{q} \text{ after } \sigma) \subseteq \{q, q_\delta\} \) for some \( q \in \mathbb{P}(S) \). So, \( (\bar{q} \text{ after } \sigma) \) is either \( \{q, q_\delta\}, \{q\} \) or \( \{q_\delta\} \). Therefore, it is sufficient to show that \( q_\delta \in \)
4.2. The Effect of Models on the Power of Testing

\((\bar{q} \text{ after } \sigma)\) implies \(q \in (\bar{q} \text{ after } \sigma)\). To this end, we assume that \(q_\delta \in (\bar{q} \text{ after } \sigma)\), i.e., \(\bar{q} \overset{\sigma}{\Rightarrow} q_\delta\). We know that \(q_\delta\) is reachable only under Rule \(R_2\). Therefore, \(q_\delta\) is reached only from state \(q\) by an internal transition or a sequence of \(\delta\)'s; thus, \(\bar{q} \overset{\sigma'}{\Rightarrow} q \overset{\delta}{\Rightarrow} q_\delta\) where \(\sigma' \delta \in \delta^*\) and \(\sigma = \sigma' \sigma_\delta\). Following \(\sigma \in \mathcal{L}_\delta^*\mathcal{L}\), we obtain that 
\[\sigma_\delta = \epsilon.\] 
Thus, \(\sigma' = \sigma\), and therefore \(q \in \text{out}(\bar{q} \text{ after } \sigma)\) which was to be shown. So, \((\bar{q} \text{ after } \sigma)\) is either \(\{q, q_\delta\}\) or \(\{q\}\) for some \(q \in \wp(S)\).

\[\Box\]

Figure 4.3: (a) The suspension automaton \(\bar{s}\) of the input-output labeled transition system \(s\) depicted in Figure 4.2, (b) The internal-choice input-output labeled transition system \(\bar{q}\) over the sets of inputs \(\{a\}\) and outputs \(\{x, \sigma\}\) that is constructed according to the interference rules of Proposition 4.7 from IOLTS specification \(s\) in Figure 4.2.

The example below illustrates how an internal-choice model is derived from an IOLTS specification according to Proposition 4.7.

**Example 4.8.** Consider the LTSs, depicted in Figure 4.3. The left-hand side LTS is the suspension automaton \(\bar{s}\) of IOLTS specification \(s\) pictured in Figure 4.2. IOLTS\(\uparrow \bar{q}\) is constructed from LTS \(\bar{s}\) according to the inference rules of Proposition 4.7. Consider the initial state of IOLTS\(\uparrow \bar{q}\) that equals \(\{\bar{s}\}\). Since observing quiescence is allowed at the initial state \(\bar{s}\), a \(\tau\)-labeled transition, according to inference Rule \(R_2\), is added at the state \(\bar{q}\) to the quiescent state \(q_1\) where \(q_1 = \bar{q}_\delta\). Because observing quiescence is possible at the initial state \(\bar{s}\) before executing the input transition \(a\), under the inference Rule \(R_3\), the transition \(\bar{q}_\delta \overset{a}{\Rightarrow} q_2\) is added to IOLTS\(\uparrow \bar{q}\) where \(q_2 = \{s_1, s_2\}\). Since all states included in \(q_2\) allow for observing quiescence, the transition \(q_2 \overset{\tau}{\Rightarrow} q_3\) is added to the transition relation of IOLTS \(\bar{q}\). Subsequently, due to inference Rule \(R_3\) the input transition \(a\) is added at state \(q_3\). Thus, the transition \(q_3 \overset{a}{\Rightarrow} q_4\) is added where \(q_4 = \{s_3, s_4, s_5\}\). Obviously, the two states \(s_3\) and \(s_5\) do not produce any common output. So, the \(\sigma\)-labeled self-loop is added in state \(q_4\) due to the inference Rule \(R_4\).

The proposition given below establishes a connection between a state in the constructed internal-choice model according to Proposition 4.7 and its corresponding states in the original IOLTS specification. More precisely, it shows that all states collected in a single state are reachable via traces with the same sequence of input and output actions.
Before formally establishing this connection, we first define the operator $\_\sigma$ on a suspension trace $a\sigma$ and a subset $X$ of $L_\delta$ as $\epsilon_{\_\sigma} = \epsilon$ and $(a\sigma)_{\_\sigma} = a\sigma$ if $a \in X$, otherwise $(a\sigma)_{\_\sigma} = \sigma_{\_\sigma}$.

**Proposition 4.9.** Let $L = I \cup U$, and let $(S, L, \rightarrow_s, \bar{s})$ be a suspension automaton of an IOLTS. Let $(Q, I, U \cup \{\sigma\}, \rightarrow, \bar{q})$ be an IOLTS$^\tau$ which is constructed according to Proposition 4.7 from LTS $\bar{s}$. Let $q \in \mathbb{P}(S)$ be reachable from the initial state $\bar{q}$ with a sequence $\sigma \in L_\delta^+$, i.e., $\bar{q} \Rightarrow q$.

Then, for all $s \in q$, there exists a sequence $\sigma' \in L_\delta^+$ such that $\bar{s} \delta_{\sigma} \rightarrow_s s$ and also $\sigma_{\_\sigma} = \sigma'_{\_\sigma}$.

**Proof.** The proof is given by induction on the length of the sequence $\sigma$.

- **Base case.** For the base case, we suppose that the length of $\sigma$ is zero, i.e., $\sigma = \epsilon$. Thus, $\bar{q} \Rightarrow q$. We distinguish two cases regarding $q$: $q = \bar{q}$ and $q \neq \bar{q}$. Since $\bar{q} \in \{\bar{s}\}$, the above proposition holds for the case $q = \bar{q}$ for $\sigma' = \epsilon$. We next consider $q \neq \bar{q}$. Following inference rules in Proposition 4.7, the transition can be taken as a consequence of deduction Rule $R_2$. Therefore, it is obtained that $q = \bar{q}_\delta$ which contradicts $q \in \mathbb{P}(S)$. Hence, $q \neq \bar{q}$ is not the case.

- **Induction step.** We assume for the induction hypothesis that the above proposition holds for all sequences in $L_\delta^+$ with the length of at most $n - 1$ and that the length of $\sigma$ is $n$. We take $\sigma = \rho a$ with $\rho \in L_\delta^+$ and $a \in L_\delta$. Thus, the length $\rho$ is $n - 1$. We distinguish three cases based on $a$.

  - We suppose that $a = \delta$. Following the second item in Proposition 4.7, there exists a state $q' \in \mathbb{P}(S)$ such that $(\bar{q} \text{ after } \rho) \subseteq \{q', q''_\rho\}$. Regarding the inference rules in Proposition 4.7, more precisely, Rule $R_2$, we know that the set of states $\{q', q''_\rho\}$ after executing a $\delta$-labeled transition reaches to the set $\{q''_\rho\}$. It is therefore obtained that $(\bar{q} \text{ after } \rho \delta) = \{q''_\rho\}$. Thus, $q = q''_\rho$ which contradicts $q \in \mathbb{P}(S)$. Hence, $a = \delta$ is not the case.

  - We suppose that $a \in U$. We know from the second item in Proposition 4.7 that $(\bar{q} \text{ after } \rho) \subseteq \{q', q''_\rho\}$ for some $q' \in \mathbb{P}(S)$. Combining with that $a \in \text{out}(\bar{q} \text{ after } \rho)$ yields $q' \in (\bar{q} \text{ after } \rho)$ because $q''_\rho$ is quiescent and cannot perform any output transition. Thus, it is obtained that $q \Rightarrow q' \Rightarrow q$. We know there exist two states $q'_1$ and $q''_1$ such that $q' \overset{\epsilon}{\Rightarrow} q'_1 \overset{\alpha}{\Rightarrow} q''_1 \overset{\epsilon}{\Rightarrow} q$. Regarding inference rules in Proposition 4.7, a $\tau$-labeled transition is taken as a consequence of deduction Rule $R_2$ that leads to some quiescent state $r_\delta$ for some $r \in \mathbb{P}(S)$ while $r_\delta \not\in \mathbb{P}(S)$ and $r \overset{\tau}{\rightarrow} r_\delta$. Consequently, $q'_1 = q$ is obtained, since state $q'_1$ is not quiescent because of the output action $a$. Moreover, we obtain that $q = q''_2$, because $q \in \mathbb{P}(S)$. Thus, $q' \overset{\alpha}{\Rightarrow} q$. The output transition $q' \overset{\alpha}{\Rightarrow} q$ is a consequence of deduction Rule $R_1$, thus $q = \{s | s' \in q', s' \overset{\alpha}{\rightarrow}_s s\}$. We assume an arbitrary state $s \in q$ and prove the above thesis for that arbitrary state. We know from the last observation that there exists a state $s' \in q'$ such that $s' \overset{\alpha}{\rightarrow}_s s$. Following the induction hypothesis, there exists a $\rho' \in L_\delta^+$ such that $\rho'_{\_\sigma} = \rho_{\_\sigma}$ and $\bar{s} \overset{\rho'}{\rightarrow}_s s'$. Therefore, $\bar{s} \overset{\rho'}{\rightarrow}_s s' \overset{\alpha}{\rightarrow}_s s$ is obtained. On the other hand, it is concluded from the projection definition that $(\rho' a)_{\_\sigma} = (\rho a)_{\_\sigma}$. Hence, the sequence $\sigma' = \rho'x$ fulfills the two conditions of the above thesis.
We show in the proposition below that the testing power of the IOLTS \( \bar{\rho} \) is inferior to the testing power of the original specification \( \hat{\rho} \). More formally, the set of correct implementations of IOLTS \( \hat{s} \) with respect to the transition relation \( \text{ioco} \) is a subset of the set of conforming implementations of IOLTS \( \bar{\rho} \).

**Proposition 4.10.** Let LTS \( \hat{s} \) be a suspension automaton of an IOLTS specification, and let IOTS \( \bar{i} \) be an input-enabled implementation, i.e., \( \bar{i} \in \text{IOTS}^{\pi}(I,U) \cap \text{IOTS}(I,U) \). Let IOLTS \( \bar{\rho} \)
\( \tilde{q} \) be an internal-choice specification which is derived from LTS \( \bar{s} \) according to the inference rules in Proposition 4.9. Then,

\[
\tilde{i} \ioco \bar{s} \text{ implies } \tilde{i} \ioco \tilde{q}
\]

**Proof.** We prove the contrapositive version of the above thesis, i.e., \( \tilde{i} \ioco \tilde{q} \) implies that \( \tilde{i} \ioco \tilde{s} \). We know from \( \tilde{i} \ioco \tilde{q} \) that there exist a sequence \( \sigma \in \Straces(\tilde{q}) \) and an output \( x \in U \cup \{ \delta \} \) such that \( x \in \text{out}(\tilde{i} \text{ after } \sigma) \) but \( x \notin \text{out}(\tilde{q} \text{ after } \sigma) \). Note that the output action \( \sigma_{\tilde{q}} \) is never produced by the implementation \( \tilde{i} \). It is consequently obtained that \( \sigma \in L^x_0 \cap \Straces(\tilde{q}) \). We also observe that the sequence \( \sigma \) does not end with a \( \delta \), i.e., \( \sigma \in \Straces(\tilde{q}) \cap (L^x_0 L) \), because otherwise both \( \tilde{i} \) and \( \tilde{q} \) get to quiescent states after executing the sequence \( \sigma \) where \( \text{out}(\tilde{i} \text{ after } \sigma) = \text{out}(\tilde{q} \text{ after } \sigma) = \{ \delta \} \). We distinguish two cases; \( \sigma = \epsilon \) and \( \sigma \in L^x_0 L \).

- We suppose that \( \sigma = \epsilon \). We know that \( \hat{q} \in (\hat{q} \text{ after } \epsilon) \). Thus, \( x \notin \text{out}(\hat{q} \text{ after } \epsilon) \) implies that \( x \notin \text{out}(\hat{q}) \). Subsequently, it follows from inference Rules \( R_1 \) and \( R_2 \) that \( x \notin \text{out}(\hat{s}) \), because \( \hat{q} = \{ \hat{s} \} \). Therefore, we obtain that \( \tilde{i} \ioco \tilde{s} \), because \( \text{out}(\tilde{i} \text{ after } \epsilon) \notin \text{out}(\hat{s} \text{ after } \epsilon) \).

- We suppose that \( \sigma \in L^x_0 L \). As a direct consequence of \( \tilde{i} \in \IOTSS(I, U) \), we have for any sequence \( \sigma' \in L^x_0 L \) where \( \sigma'_{\tilde{i}} = \sigma_{\tilde{q}} \) that \( \tilde{i} \text{ after } \sigma' = \tilde{i} \text{ after } \sigma \); thus \( \text{out}(\tilde{i} \text{ after } \sigma') = \text{out}(\tilde{i} \text{ after } \sigma) \). To prove the above thesis, therefore, it is sufficient to show that there exists a state \( s \in S \) such that \( x \notin \text{out}(s) \) and also \( s \sigma'_{\tilde{i}} \rightarrow s' \) for \( \sigma' \in L^x_0 L \) such that \( \sigma'_{\tilde{i}} = \sigma_{\tilde{q}} \). We take \( \sigma = \rho a \), where \( \rho \in L^x_0 \) and \( a \in L \). We distinguish two cases; \( a \in U \) and \( a \in I \).

  - We suppose that \( a \in U \). It is clear that there exist two states \( q', q \in Q \) such that \( \hat{q} \xrightarrow{\rho} q' \xrightarrow{a} q \). Following that an output transition in \( \IOTSS \hat{q} \) is a consequence of inference Rule \( R_1 \), we find that \( q' \in \mathcal{P}(S) \) as well as \( q = \{ s \mid s' \in q', s' \xrightarrow{a} s \} \).

  - We suppose that \( a \in I \). Clearly, there exist two state \( q', q \in Q \) such that \( \hat{q} \xrightarrow{\rho} q' \xrightarrow{a} q \). Following that an input transition in \( \IOTSS \hat{q} \) is a consequence of inference Rule \( R_3 \), there must exist a state \( q'' \in \mathcal{P}(S) \) such that \( q'' = q''_{\tilde{q}} \) and \( q = \{ s \mid s' \in q'', s' \xrightarrow{\delta a} s \} \). We know from the inference rules in Proposition 4.7 that \( q''_{\tilde{q}} \) is reachable only under deduction Rule \( R_2 \) via state \( q'' \) by an internal transition or a sequence of \( \delta \) actions. Thus, the transition \( \hat{q} \xrightarrow{\rho'} q'' \xrightarrow{\rho \delta} q' \xrightarrow{a} q \) is obtained where \( \rho = \rho' \sigma_{\tilde{q}} \) with \( \sigma_{\tilde{q}} \in \delta^* \). The fact that \( q \in (\hat{q} \text{ after } \sigma) \) yields that \( \text{out}(q) \subseteq \text{out}(\hat{q} \text{ after } \sigma) \); thus, \( x \notin \text{out}(q) \). With respect to the inference
4.2. The Effect of Models on the Power of Testing

rules $R_1$ and $R_2$, it is concluded from the last result that there exists a state $s \in q$ such that $x \not\in \text{out}(s)$. We also know that there exists a state $s' \in q''$ such that $s' \stackrel{\delta^*a}{\rightarrow} s$. We know from Proposition 4.9 that there exists a sequence $\rho'' \in L^*_\delta$ such that $\bar{s} \stackrel{\rho''}{\rightarrow} s''$ and $\rho'' \downarrow \sigma \downarrow s$. Therefore, state $s$ is reachable from the initial state $\bar{s}$ by the sequence $\sigma' = \rho'' \delta^*a$, i.e., $\bar{s} \stackrel{\sigma'}{\rightarrow} s$. It is obtained from the definition of the projection operator that $\sigma' \downarrow s = \sigma \downarrow s$.

So far, we have shown that there exists a state $s \in S$ such that $x \not\in \text{out}(s)$ and also $\bar{s} \Rightarrow s$ for $\sigma' \in L^*_\delta$ where $\sigma' \downarrow s = \sigma \downarrow s$. On the other hand, we deduce from $\text{out}(\bar{i} \text{ after } \sigma') = \text{out}(\bar{i} \text{ after } \sigma)$ that $x \in \text{out}(\bar{i} \text{ after } \sigma')$. Combining the last two observations results in $\text{out}(\bar{i} \text{ after } \sigma') \not\subseteq \text{out}(\bar{s} \text{ after } \sigma')$; thus, $\text{io}_\text{co} \bar{s}$ which was to be shown.

$\square$

To show the $\text{io}_\text{co}$-testing equivalence of the IOLTS $\bar{q}$ to the original given IOLTS $\bar{s}$, we need to show that the discriminating power of IOLTS $\bar{q}$ is finer than IOLTS $\bar{s}$. In more formal words, we have to show that all incorrect implementations of IOLTS $\bar{q}$ with respect to the $\text{io}_\text{co}$ relation do not conform to IOLTS $\bar{s}$ is used as the specification, which is proved through the next two propositions.

**Proposition 4.11.** Let IOLTS $\bar{q}$ be an internal-choice IOLTS which is derived from a valid suspension automaton $\bar{s}$ according to the inference rules in Proposition 4.7, and let sequence $\sigma$ be a suspension trace of $\bar{q}$ such that $\sigma \in \text{Straces}(\bar{s}) \cap (L^*_\delta L)^*$. Let $\bar{s} \Rightarrow_s s$ for $s \in S$, and let $\bar{q} \Rightarrow q$ for $q \in \mathbb{P}(S)$. Then, $s \in q$ always holds.

**Proof.** The proof is given by induction on the length of $\sigma$.

- **Base case.** We assume for the base case that $\sigma = \epsilon$. We know that $(\bar{q} \text{ after } \epsilon) = \{\bar{q}, \bar{q}_\epsilon\}$. Regarding $\bar{q} \in (\bar{q} \text{ after } \epsilon)$, the last observation yields that $(\bar{q} \text{ after } \epsilon) \subseteq \mathbb{P}(S)$ is the singleton set $\{\bar{q}\}$. Therefore, the thesis holds for $\sigma = \epsilon$, because $\bar{s} \Rightarrow_s \bar{s}$ and $\bar{q} = \{\bar{s}\}$.

- **Induction step.** We assume for the induction hypothesis that the above thesis holds for all sequences with length of less than or equal to $n-1$, and that the length of $\sigma$ is $n$; thus $\sigma \in L^*_\delta L$. We take $\sigma = \sigma' \gamma$ where $\sigma' \in L^*_\delta$ and $\gamma \in L$; thus, the length of $\sigma'$ is $n-1$. We know from the second item in Proposition 4.7 that $(\bar{q} \text{ after } \sigma) \subseteq \{q, q_\delta\}$ for some $q \in \mathbb{P}(S)$. Following the third item of Proposition 4.7, it is obtained that $(\bar{q} \text{ after } \sigma) \cap \mathbb{P}(S) = \{q\}$. To prove the above thesis, therefore, we need to show that $s \in q$ with $s \in (\bar{s} \text{ after } \sigma)$. We distinguish two cases; $y \in U$ and $y \in I$.

- We suppose that $y \in U$. From $\sigma' \gamma \in \text{Straces}(\bar{q})$, we obtain that there exists a state $q' \in Q$ such that $\bar{q} \stackrel{\sigma' \gamma}{\Rightarrow} q' \Rightarrow q$. Similarly, there must exist a state $s' \in S$ such that $\bar{s} \stackrel{\sigma'}{\rightarrow} s' \gamma' \Rightarrow_s s$. The sequence $\sigma'$ cannot end in $\delta$, because $y \in U$; thus $\sigma' \in (L^*_\delta L)^*$. It is also observed that the state $q'$ is not quiescent because of the presence of an output action $y$ that subsequently leads to $q' \in \mathbb{P}(S)$.  

59
Following the induction hypothesis, it is obtained that \( s' \in q' \). We know from the transition \( q' \xrightarrow{\gamma} q \) that there exist two states \( q_1'' \), \( q'' \in Q \) such that \( q' \xrightarrow{\delta} q_1'' \xrightarrow{\gamma} q'' \xrightarrow{\epsilon} q \). Since \( q_1'' \) is not quiescent, it is deduced with respect to inference Rule \( R_2 \) that \( q' = q'' \). It is concluded from \( q \in \mathcal{P}(S) \) that \( q'' = q \), because otherwise \( q = q'' \) is obtained regarding inference Rule \( R_2 \) where \( q'' \notin \mathcal{P}(S) \). Summarizing the above lines, we observe that \( q' \xrightarrow{\gamma} q \). Since an output transition is only a consequence of inference Rule \( R_1 \), \( q = \{ s \mid s' \in q', s' \xrightarrow{\gamma} s \} \) is obtained. Combining the last observation with \( s' \in q' \) results in \( s \in q \) which was to be shown.

We suppose that \( y \in I \). We break down \( \sigma' \) into two subsequences such that \( \sigma' = \sigma'' \sigma_\delta \) where \( \sigma'' \in (L_\delta L)^* \) and \( \sigma_\delta \in \delta^* \); thus \( \sigma = \sigma'' \sigma_\delta y \). Consequently, we know from \( \sigma \in \text{Straces}(\bar{q}) \) that there exist two states \( q', q_1' \in Q \) such that \( \bar{q} \xrightarrow{\sigma''} q' \xrightarrow{\sigma_\delta} q_1' \xrightarrow{\gamma} q \). Likewise, there exist two states \( s', s_1' \in S \) such that \( s \xrightarrow{\sigma''} s' \xrightarrow{\sigma_\delta} s_1' \xrightarrow{\gamma} s \).

We first show that we can always find \( q' \in Q \) such that \( q' \in \mathcal{P}(S) \). In this regard, we assume that \( q' \notin \mathcal{P}(S) \), and then show there is a state \( q'' \in \mathcal{P}(S) \) such that \( \bar{q} \xrightarrow{\sigma''} q'' \xrightarrow{\epsilon} q' \). Following inference Rule \( R_2 \), \( q' \) is only available from a state \( q'' \in \mathcal{P}(S) \) such that \( q' = q'' \) via internal transitions or a sequence of \( \delta \). The last observation together with \( \sigma'' \in (L_\delta L)^* \) results in \( \bar{q} \xrightarrow{\sigma''} q'' \xrightarrow{\epsilon} q' \).

Therefore, in the transition \( \bar{q} \xrightarrow{\sigma''} q'' \xrightarrow{\epsilon} q' \xrightarrow{\gamma} q \) we can always take \( q' \in \mathcal{P}(S) \). Following the induction hypothesis, \( s' \in q' \) is obtained. On the other hand, it is concluded from \( \sigma_\delta \in \delta^* \) with respect to Rule \( R_2 \) that \( q_1' = q'' \). There, consequently, exists a state \( s'' \in \mathcal{P}(S) \) such that \( s'' \xrightarrow{\gamma} q'' \xrightarrow{\epsilon} q \). Following \( q \in \mathcal{P}(S) \), we get \( q = q'' \), because otherwise inference Rule \( R_2 \) results in \( q = q'' \) where \( q_1'' \notin \mathcal{P}(S) \). Therefore, we have \( q' \xrightarrow{\sigma_\delta} q'' \xrightarrow{\gamma} q \). Since an input transition is a consequence of inference Rule \( R_3 \), \( q = \{ s \mid s' \in q', s' \xrightarrow{\gamma} s \} \) is obtained. Combining of the last observation with \( s' \in q' \) leads to \( s \in q \) which was to be shown.

\[ \square \]

**Proposition 4.12.** Let LTS \( \bar{s} \) be a valid suspension automaton, and let IOTS\( ^\forall \bar{t} \) be an input-enable implementation, i.e., \( \bar{t} \in \text{IOTS}^\forall(I, U) \cup \text{IOTS}(I, U) \). Let IOLTS \( \bar{q} \) be an internal-choice specification which is derived from LTS \( \bar{s} \) according to the inference rules in Proposition 4.7. Then,

\[ \bar{t} \text{ioco} \bar{q} \text{ implies } \bar{t} \text{ioco} \bar{s} \]

**Proof.** We prove the contrapositive version of the above thesis, i.e., \( \bar{t} \text{ioco} \bar{s} \) implies that \( \bar{t} \text{ioco} \bar{q} \). We know from \( \bar{t} \text{ioco} \bar{s} \) that there exists a sequence \( \sigma \in \text{traces}(\bar{s}) \) and an output \( x \in U \cup \{ \delta \} \) such that \( x \notin \text{out}(\bar{t} \text{ after } \sigma) \) but \( x \notin \text{out}(\bar{s} \text{ after } \sigma) \). Note that the sequence \( \sigma \) does not end with \( \delta \), i.e., \( \sigma \in \text{traces}(\bar{s}) \cap (L_\delta L) \), because otherwise \( \text{out}(\bar{t} \text{ after } \sigma) = \text{out}(\bar{q} \text{ after } \sigma) = \{ \delta \} \) is deduced, which contradicts \( \text{out}(\bar{t} \text{ after } \sigma) \notin \text{out}(\bar{q} \text{ after } \sigma) \). We distinguish two cases; \( \sigma \in \text{Straces}(\bar{q}) \) and \( \sigma \notin \text{Straces}(\bar{q}) \).
4.2. The Effect of Models on the Power of Testing

- We suppose that \( \sigma \in \text{Straces}(\bar{q}) \); thus \( \bar{q} \) after \( \sigma \) \( \neq \emptyset \). We know from the second item in Proposition 4.7 that \( \text{out}(\bar{q} \text{ after } \sigma) \subseteq \{q', q^*_3\} \) for some \( q' \in P(S) \). Therefore, \( \bar{q} \) after \( \sigma \) is either \( \{q', q^*_3\}, \{q'\} \) or \( \{q^*_3\} \). Consequently, it follows from the third item of Proposition 4.7 that \( \bar{q} \) after \( \sigma \) \( \neq \{q^*_3\} \). Thus, \( q' \in \bar{q} \text{ after } \sigma \). Following Proposition 4.11, we can find a \( \bar{q} \) after \( \sigma \). Consequently, it is obtained from \( \bar{q} \) after \( \sigma \) that \( \sigma \) in \( \bar{q} \text{ after } \sigma \). We first show that \( q \in \bar{q} \text{ after } \sigma \). We prove the above thesis by showing that \( \delta \in \bar{q} \text{ after } \sigma \). We take \( \sigma' = \sigma'' \sigma_\delta \) where \( \sigma'' \in (L_\delta^+ L)^* \) and \( \sigma_\delta \in \delta^+ \). Therefore, there exist two states \( s, s' \in S \) such that \( s \stackrel{\sigma''}{\rightarrow} s', s \stackrel{\sigma_\delta}{\rightarrow} s \). Similarly, there are states \( q, q' \in Q \) such that \( q \stackrel{\sigma''}{\rightarrow} q \stackrel{\sigma_\delta}{\rightarrow} q' \). Using a similar line of reasoning as in the proof of Proposition 4.11, we can find a \( q \in Q \) such that \( q \in P(S) \). Consequently, it is obtained that \( q \in \bar{q} \text{ after } \sigma \). To this end, we distinguish two cases; \( \sigma_\delta = \epsilon \) or \( \sigma_\delta \in \delta^+ \).

  * We suppose that \( \sigma_\delta = \epsilon \). It is concluded from \( \bar{q} \notin \text{Straces}(q') \) that \( y \notin \text{Straces}(q') \) either.
  * We suppose that \( \sigma_\delta \in \delta^+ \). We know from the set of inference rules in Proposition 4.7 that \( \{q \text{ after } \sigma_\delta\} = \{q_\delta\} \); thus \( q' = q_\delta \). It is known from Rule R_3 that \( q_\delta \) is reachable from \( q \) by an internal transition. Combining the inference rule R_2 with \( y \notin \text{Straces}(q') \) yields that \( y \notin \text{Straces}(q) \).

Towards a contradiction with \( y \notin \text{Straces}(q) \), we assume that \( \delta \in \text{out}(\bar{q}) \). Regarding the inference rules of Proposition 4.7, more specifically Rule R_2, it is obtained that \( q \stackrel{\sigma''}{\rightarrow} q_\delta \) when \( \forall s \in q \bullet s \stackrel{\delta}{\rightarrow} s \). It is, consequently, concluded from Rule R_3 due to \( s \in q \) that \( q_\delta \rightarrow; \) thus, \( y \in \text{Straces}(q) \). Therefore, \( \delta \notin \text{out}(\bar{q}) \) is proven. Consequently, it is obtained that \( \bar{q} \text{ after } \sigma'' \) = \( \{q\} \); thus, \( \delta \notin \text{out}(\bar{q} \text{ after } \sigma'') \). On the other hand, we know that \( \delta \in \text{out}(\bar{i} \text{ after } \sigma'') \), because implementation \( \bar{i} \) is both an input-enabled and internal-choice IOLTS, i.e., \( \bar{i} \in \text{IOTS}(I, U) \cap \text{IOTS}^\cap(I, U) \). The last two observations result in \( \bar{i} \text{ downco } \bar{q} \) which was to be shown.

The above propositions prove that for every IOLTS specification \( \bar{s} \), an \textbf{iooco}-testing equivalent IOLTS\(^\cap \) \( \bar{q} \) can be constructed. The language of IOLTS\(^\cap \) \( \bar{q} \), nevertheless, is not the same as the language of IOLTS \( \bar{s} \), because of the fresh output \( \sigma \) which presents the absence of all other outputs including quiescence. In order to prove the equality of the testing power of IOLTSs and IOLTS\(^\cap\)'s over the IOTS\(^\cap \) input-enabled implementations, we
still need to find an internal-choice specification over the same sets of inputs and outputs, i.e., over the language $L$. In the remainder of this section, we present a transformation function from IOLTS $\tilde{q}$ with the language of $L$ to a testing-equivalent IOLTS whose language is $L$.

We observe that the only source of nondeterminism in IOLTS $\tilde{q}$ derived from an IOLTS specification according to Proposition 4.7 is internal transitions leading to quiescent states. Figure 4.4 illustrates the structural form of transitions of IOLTS $\tilde{q}$. Before introducing the transformation function on IOLTS $\tilde{q}$, we first formally characterize those structural properties of IOLTS $\tilde{q}$ on internal-choice IOLTSs. It will be shown that these properties are preserved during the transformation.

Our transformation relies on the observation that no output but $\sigma$ is allowed in IOLTS $\tilde{q}$ after reaching a state where the $\sigma$-labeled transition is enabled. Therefore, all incoming transitions to such states can be safely removed with no effect on the power of testing, which will be shown in the propositions below. Before giving the formal definition of the transformation, we first define $\leq_S \subseteq S \times S$ as an arbitrary order on the set of states $S$, and subsequently $<_I$ to be the lexicographical order on a vector of $S$. 

---

**Figure 4.4:** A structurally bounded nondeterministic IOLTS $\tilde{s}$, where (a) shows the structural form of the input transition $a$ which is deterministically taken at a quiescent state (b) depicts the structural form of the output transition $x$ which is deterministically taken, and (c) demonstrates a $\sigma$-enabled state

**Definition 4.13.** Let $\langle S, I, U \cup \{\sigma\}, \rightarrow, \tilde{s} \rangle$ be an IOLTS, and let $L = I \cup U$. We say $\tilde{s}$ is structurally bounded nondeterministic (SBN) iff it has the following properties:

1. $\forall s \in \text{der}(\tilde{s}) \bullet \sigma \in \text{init}(s)$ implies $\text{init}(s) = \{\sigma\} \land s \xrightarrow{\sigma} s$
2. $\forall s, s' \in \text{der}(\tilde{s}) \bullet s \xrightarrow{\tau} s'$ implies $\delta(s')$
3. $\forall s \in \text{der}(\tilde{s}) \bullet \delta(s)$ implies $\exists s' \in \text{der}(\tilde{s}) \bullet s' \xrightarrow{\tau} s$
4. $\forall s \in \text{der}(\tilde{s}), \forall a \in \text{init}(s) \cap L, \forall s' \in \text{der}(\tilde{s}) \bullet a \xrightarrow{s'}$ implies $\neg \delta(s')$
5. $\forall s \in \text{der}(\tilde{s}), \forall a \in \text{init}(s), \forall s', s'' \in \text{der}(\tilde{s}) \bullet a \xrightarrow{s'} s' \land a \xrightarrow{s} s''$ implies $s' = s''$

Our transformation relies on the observation that no output but $\sigma$ is allowed in IOLTS $\tilde{q}$ after reaching a state where the $\sigma$-labeled transition is enabled. Therefore, all incoming transitions to such states can be safely removed with no effect on the power of testing, which will be shown in the propositions below. Before giving the formal definition of the transformation, we first define $\leq_S \subseteq S \times S$ as an arbitrary order on the set of states $S$, and subsequently $<_I$ to be the lexicographical order on a vector of $S$. 

62
4.2. The Effect of Models on the Power of Testing

Definition 4.14. Let \( \langle S, I, U \cup \{ \sigma \}, \rightarrow, \bar{s} \rangle \) be an SBN IOLTS\( ^\upsilon \), and let \( B_i(\bar{s}) \) be defined as \( \{(s, s', s'') \in \text{der}(\bar{s}) \mid \exists a \in I : s \overset{a}{\rightarrow} s' \overset{\sigma}{\rightarrow} s'' \} \). Then, the transformation \( T_1 : \text{IOLTS}(I, U \cup \{ \sigma \}) \rightarrow \text{IOLTS}(I, U \cup \{ \sigma \}) \) is defined as \( T_1(\bar{s}) = \langle Q, I, U \cup \{ \sigma \}, \rightarrow_t, \bar{q} \rangle \), where \( Q = S \), \( \bar{q} = \bar{s} \), and \( \rightarrow_t \) is deduced from \( \rightarrow \) as follows:

\[
\rightarrow_t = \begin{cases} 
\rightarrow \setminus \{(s, \tau, s'), (s', a, s'') \mid (s, s', s'') \text{ is the smallest triple in } B_i(\bar{s}) \} & \text{if } B_i = \emptyset \\
\cup \{(s, \sigma, s) \mid (s, s', s'') \text{ is the smallest triple in } B_i(\bar{s}) \} & \text{otherwise} 
\end{cases}
\]

After executing an input transition leading to a \( \sigma \)-enabled state, the IUT will, regardless of its outputs, always be recognized as a non-conforming implementation. Therefore in testing, we do not need to proceed beyond such an input transition. In this sense, as stated in Definition 4.14, Transformation \( T_1 \), an input transition along with with its preceding \( \tau \)-labeled transition can be removed. Furthermore, the removal of those transitions, as illustrated in Figure 4.5, preserves the structural properties of a given SBN IOLTS\( ^\upsilon \) specification, given in Definition 4.13; thus, the transformed specification remains an SBN IOLTS\( ^\upsilon \).

![Figure 4.5: The effect of Transformation \( T_1 \)](image)

We prove in the proposition given below that the power of identifying of incorrect implementations is not affected by transformation \( T_1 \).

Proposition 4.15. Let \( \langle S, I, U \cup \{ \sigma \}, \rightarrow, \bar{s} \rangle \) be an SBN IOLTS\( ^\upsilon \), and let \( T_1(\bar{s}) = \langle Q, I, U \cup \{ \sigma \}, \rightarrow_t, \bar{q} \rangle \). Then,

1. \( T_1(\bar{s}) \) is an SBN IOLTS\( ^\upsilon \).

2. For any implementation \( \bar{r} \in \text{IOTS}(I, U) \cap \text{IOTS}\( ^\upsilon \)(I, U) \), we have

\( \bar{r} \text{ ioco } \bar{s} \iff \bar{r} \text{ ioco } \bar{q} \)

Proof. We consider each item separately. The first item is illustrated in Figure 4.5. Now, we prove the contrapositive version of each implication of the second item.
• We show that \( \bar{i}q\bar{o}q \) implies \( \bar{i}q\bar{o}q \). From \( \bar{i}q\bar{o}q \), we obtain that for a sequence \( \sigma \in \text{Straces}(\bar{s}) \), \( \text{out}(\bar{r} \text{ after } \sigma) \not\subseteq \text{out}(\bar{s} \text{ after } \sigma) \). Note that \( \sigma \) does not end in \( \delta \), because otherwise \( \text{out}(\bar{r} \text{ after } \sigma) = \text{out}(\bar{s} \text{ after } \sigma) = \{\delta\} \). We distinguish two cases; \( \sigma \in \text{Straces}(\bar{q}) \) and \( \sigma \not\in \text{Straces}(\bar{q}) \).

  - We suppose that \( \sigma \in \text{Straces}(\bar{q}) \). We know that \( (\bar{q} \text{ after } \sigma) \subseteq \{q, q_0\} \) for some non-quiescent \( q \in S \) such that \( q \xrightarrow{\bar{a}} q_0 \). Since \( \sigma \in L_q^s \), we obtain using a similar line of reasoning as of the proof of the third item in Proposition 4.7, that \( q \in (\bar{s} \text{ after } \sigma) \). Moreover, we have due to transformation \( T_1 \) that \( q \in (\bar{q} \text{ after } \sigma) \). We now make a distinction between the following two cases: \( q \neq s \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \), and \( q = s \).

    * We assume that \( q \neq s \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \). Thus Transformation \( T_1 \) dose not change the set of outgoing transitions of state \( q \), i.e., \( \forall a \in U \cup \{\delta, \bar{a}\} \bullet q \xrightarrow{a} \text{ if and only if } q \xrightarrow{\bar{a}} \). Therefore, since IOLTS\( ^{1}_{\text{I}} \), \( \bar{q} \) is an SBN, \( \text{out}(\bar{s} \text{ after } \sigma) = \text{out}(\bar{q} \text{ after } \sigma) \) is obtained which results in \( \bar{i}q\bar{o}q \).

    * We assume that \( q = s \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \). We deduce from transformation \( T_1 \) that \( \text{out}(\bar{q} \text{ after } \sigma) \subseteq \text{out}(\bar{s} \text{ after } \sigma) \). Therefore, we obtain that \( \text{out}(\bar{r} \text{ after } \sigma) \not\subseteq \text{out}(\bar{q} \text{ after } \sigma) \); thus \( \bar{i}q\bar{o}q \).

  - We suppose that \( \sigma \not\in \text{Straces}(\bar{q}) \). We know from transformation \( T_1 \) that the set of differences of \( \text{Straces}(\bar{s}) \) and \( \text{Straces}(\bar{q}) \) is a subset of \( D = \{\sigma \delta a \sigma' \mid \sigma \delta \in \bar{\delta}, a \in I, \sigma' \in \text{Straces}(\bar{s})\} \), where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \). Therefore, there exist \( \sigma', \sigma'' \in L_q^s \) such that \( \sigma = \sigma' \sigma'' \) and \( \sigma'' \in D \). Following Definition 4.13 combined with the last observation, it is obtained that \( (\bar{s} \text{ after } \sigma') = \{s, s'\} \) while \( (\bar{q} \text{ after } \sigma') = \{s\} \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \). Consequently, \( \bar{\delta} \not\in \text{out}(\bar{q} \text{ after } \sigma) \). On the other hand, since \( \sigma'' \in \text{Straces}(\bar{r} \text{ after } \sigma') \), it is concluded from \( \bar{r} \in \text{IOTS}^{1}_{\text{I}}(I, U) \) that \( \bar{\delta} \in \text{out}(\bar{r} \text{ after } \sigma') \). The combination of the two last observation results in \( \bar{i}q\bar{o}q \).

• We show that \( \bar{i}q\bar{o}q \) implies \( \bar{i}q\bar{o}q \). Following \( \bar{i}q\bar{o}q \), there exists a sequence \( \sigma \in \text{Straces}(\bar{q}) \) such that \( \text{out}(\bar{r} \text{ after } \sigma) \not\subseteq \text{out}(\bar{q} \text{ after } \sigma) \). Like the previous case, note that \( \sigma \in L_q^s \). We distinguish two cases; \( \sigma \in \text{Straces}(\bar{s}) \), and \( \sigma \not\in \text{Straces}(\bar{s}) \).

  - We suppose that \( \sigma \in \text{Straces}(\bar{s}) \). Similar to the previous case, it is obtained that there exists a non-quiescent state \( q \in S \) such that \( q \in (\bar{q} \text{ after } \sigma) \), and, moreover, \( q \in (\bar{s} \text{ after } \sigma) \) due to transformation \( T_1 \). We now distinguish two cases; \( q \neq s \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \), and \( q = s \).

    * We assume that \( q \neq s \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \); thus \( \forall a \in U \cup \{\delta, \sigma\} \bullet q \xrightarrow{a} \) if and only if \( q \xrightarrow{\bar{a}} \). Therefore, \( \text{out}(\bar{q} \text{ after } \sigma) = \text{out}(\bar{s} \text{ after } \sigma) \) is obtained which results in \( \bar{i}q\bar{o}q \).

    * We assume that \( q = s \) where \( (s, s', s'') \) is the smallest triple of \( B_1(\bar{s}) \). It is a direct consequence of transformation \( T_1 \) that \( \bar{q} \) after executing \( \sigma \) produces all outputs of \( (\bar{s} \text{ after } \sigma) \) but quiescence, i.e., \( \text{out}(\bar{q} \text{ after } \sigma) = \text{out}(\bar{s} \text{ after } \sigma) \setminus \{\delta\} \). Therefore, if implementation \( \bar{r} \) is recognized as a
let Proposition 4.17. The transformed model remains unchanged compared to the IOLTS $\mathcal{I}\mathcal{O}\mathcal{L}\mathcal{T}\mathcal{S}$, as illustrated in Figure 4.6. We now consider the case that $\delta \in \text{out}(\bar{r} \text{ after } \sigma)$. We know from Transformation $T_1$ that there exists an input action $a \in I$ such that $s \xrightarrow{\delta} s' \xrightarrow{\sigma} a'' \xrightarrow{r} s''$. Since $\bar{r} \in \mathcal{I}\mathcal{O}\mathcal{L}\mathcal{T}\mathcal{S}(I, U)$, we obtain that $(\bar{r} \text{ after } \sigma a) \neq \emptyset$. Combining the last two observations result in $\bar{r} \text{ ioco } \bar{s}$. We suppose that $\sigma \in \text{Straces}(\bar{s})$. We conclude from transformation $T_1$ that $\sigma \in \bar{L}_\sigma \sigma$. We know that $\sigma \notin L$ is a fresh output symbol denoting the absence of other outputs. Thus, the last observation is in contradiction with $\sigma \in \bar{L}_\sigma \sigma$. Therefore, $\sigma \in \text{Straces}(\bar{s})$ is not the case.

\[ \square \]

Analogous to transformation $T_1$, we present below another transformation function which removes output transitions leading to $\sigma$-enabled states.

**Definition 4.16.** Let $\langle S, I, U \cup \{ \sigma \}, \rightarrow, \bar{s} \rangle$ be an SBN IOLTS, and let $B_u(\bar{s})$ be defined as $\{(s, s') \in \text{der}(\bar{s}) | s \xrightarrow{x} s' \}$. Then the transformation $T_2 : \mathcal{I}\mathcal{O}\mathcal{L}\mathcal{T}\mathcal{S}(I, U \cup \{ \sigma \}) \rightarrow \mathcal{I}\mathcal{O}\mathcal{L}\mathcal{T}\mathcal{S}(I, U \cup \{ \sigma \})$ is defined as $T_2(\bar{s}) = \langle Q, I, U \cup \{ \sigma \}, \rightarrow, \bar{q} \rangle$ where $Q = S$, $\bar{q} = \bar{s}$, and $\rightarrow$ is deduced from $\rightarrow$ as following:

$$
\rightarrow_t \begin{cases}
\rightarrow & \text{if } B_u = \emptyset \\
(\rightarrow \cup \{(s, x, s') \mid (s, s') \text{ is the smallest pair in } B_u(s)\}) \cup \{(s, \sigma, s) \mid (s, s') \text{ is the smallest pair in } B_u \land \text{init}(s) = \{x \mid x \in U, x \xrightarrow{\delta} s\}\} & \text{otherwise}
\end{cases}
$$

\[ \begin{array}{c}
\text{Transformation } T_1 \\
\text{Transformation } T_2
\end{array} \]

\[ \begin{array}{c}
\text{Figure 4.6: The effect of Transformation } T_2
\end{array} \]

Note that similar to transformation $T_1$, as illustrated in Figure 4.6, transformation $T_2$ maintains the structural properties of a given IOLTS $\bar{s}$ while the testing power of the transformed model remains unchanged compared to the IOLTS $\bar{s}$.

**Proposition 4.17.** Let $\langle S, I, U \cup \{ \sigma \}, \rightarrow, \bar{s} \rangle$ be an SBN IOLTS, and let $T_2(\bar{s}) = \langle Q, I, U \cup \{ \sigma \}, \rightarrow, \bar{q} \rangle$. Then,

1. $T_2(\bar{s})$ is an SBN IOLTS.

65
2. For any implementation \( \bar{r} \in \text{IOTS}(I, U) \cap \text{IOTS}^\Gamma(I, U) \), we have
\[
\bar{r} \text{ioco} \bar{s} \iff \bar{r} \text{ioco} \bar{q}
\]

**Proof.** The first item is straightforward as illustrated in Figure 4.6. Hence, we turn to the second item. Before its formal proof, consider the state \( s \) of IOLTS\(^\Gamma \), depicted in Figure 4.6 which is reachable via a sequence \( \sigma \in L_{\bar{s}}^* \) from the initial state. Let \( x \in U \) be an output of state \( s \) such that \( s \xrightarrow{x} s' \). Notice that state \( s' \) is a \( \sigma \)-enabled, i.e., \( \text{out}(s') = \{\sigma\} \).

We observe that upon executing the trace \( \sigma x \), IOLTS\(^\Gamma \) \( \bar{s} \) ends up at state \( s' \) at which no implementation, regardless of its outputs, is considered a correct one. Therefore, the removal of the \( x \)-labeled transition from set of outgoing edges from state \( s \) does not have any impact on the testing power of \( \bar{s} \), because for every implementation \( \bar{r} \) that can execute the trace \( \sigma x \), i.e., \((\bar{I} \text{ after } \sigma x) \neq \emptyset\), we always have \( \text{out}(\bar{r} \text{ after } \sigma x) \nsubseteq \text{out}(\bar{s} \text{ after } \sigma x) = \{\sigma\} \). Therefore, in testing with \( \bar{s} \), implementations that produces output \( x \) can safely be rejected which happens in the transformed version of \( \bar{s} \) according to transformation \( T_1 \). A \( \sigma \)-labeled self-loop is added to state \( s \), provided it does not have any outgoing transition after the removal of its outgoing transitions leading to the state \( s' \), a \( \sigma \)-enabled state, to show that no other output, including quiescence, is acceptable.

The contrapositive version of each implication of the second item is proven by similar line of reasoning as that of the proof of Proposition 4.15. \( \square \)

An incoming transition to a \( \sigma \)-enabled state is removed in each application of Transformations \( T_1 \) and \( T_2 \). The number of reachable \( \sigma \)-enabled states, therefore, will be gradually decreasing after several successive applications of transformations \( T_1 \) and \( T_2 \). Since the size of the transition relation of the original SBN IOLTS\(^\Gamma \) \( \bar{s} \) is finite, a stable transformed SBN IOLTS\(^\Gamma \) is obtained after a finite number of applications of transformations \( T_1 \) and \( T_2 \). We formally define successive applications of transformations \( T_1 \) and \( T_2 \) on the given IOLTS\(^\Gamma \) \( \bar{s} \) as the composition of those transformation functions, i.e., \((T_1 \circ T_2)(\bar{s}) = T_2(T_1(\bar{s})) \). Subsequently, \((T_1 \circ T_2)^n(\bar{s})\) with \( n \geq 1 \) denotes the \( n \)-th transformed version of IOLTS\(^\Gamma \) \( \bar{s} \) after applying the composition of transformations \( T_1 \) and \( T_2 \) for \( n \) times.

**Proposition 4.18.** Let \( \langle S, I, U \cup \{\sigma\}, \rightarrow, \bar{s} \rangle \) be an SBN IOLTS\(^\Gamma \). Then, there exists a number \( 1 \leq n < 2 * \rightarrow | \) such that,

1. \((T_2 \circ T_1)^n(\bar{s}) = (T_2 \circ T_1)^{n+1}(\bar{s})\)

2. if \((T_2 \circ T_1)^n(\bar{s})\) has a reachable \( \sigma \)-enabled state, then \( \text{init}(\bar{s}) = \{\sigma\} \).

**Proof.** Assume \( \langle S, I, U \cup \{\sigma\}, \rightarrow, \bar{s} \rangle \) is an SBN IOLTS\(^\Gamma \). Let the set \( T_0 = \{(s', a, s'') \mid s' \in \text{der}(s) \wedge (s', a, s'') \in \rightarrow \wedge a \in (I \cup U)\} \) denote the transitions of an IOLTS\(^\Gamma \) \( \langle S, I, U \cup \{\sigma\}, \rightarrow, \bar{s} \rangle \) whose labels are in \( I \cup U \cup \{\tau\} \), and let \( T_\sigma = \{(s', \sigma, s') \mid s' \in \text{der}(s) \wedge (s', \sigma, s') \in \rightarrow \} \) denote the set of the \( \sigma \)-labeled transitions of IOLTS\(^\Gamma \) \( s \). We define a measure to IOLTS\(^\Gamma \) \( s \), denoted by \#(s), defined as \#(s) = 2*\( |T_0| + |T_\sigma| \). We next show that in general the consecutive application of transformations \( T_1 \) and \( T_2 \) decreases the measure of the transformed model. More formally, we prove that \#((T_2 \circ T_1)(s)) \leq \#(s) - 1, provided \((T_2 \circ T_1)(s) \neq s \).

Since \((T_2 \circ T_1)(s) \neq s \), at least one of the transformations \( T_1 \) and \( T_2 \) is the source of changes in the transition relation of IOLTS\(^\Gamma \) \( s \). Following Definition 4.14, after applying
T₁, two transitions, an internal and an input transitions, may be removed while a new σ-labeled self-looped transition can be added. Therefore, in the case where transformation T₁ makes changes in the original transition relation, the measure of the original specification decreases by 3. Similarly in transformation T₂, an output transition may be removed as a new σ-labeled self-looped transition can be added to the transition relation. So, transformation T₂ can decrease the measure of the original specification by 1. Combining the above observations together, the application of T₂ ◦ T₁ decreases the measure of specification s by at least 1, i.e., #((T₁ ◦ T₂)(s)) ≤ #(s) − 1, if (T₂ ◦ T₁)(s) ̸= s.

A direct consequence of the last observation is #((T₁ ◦ T₂)(s)) ≤ #(s) − n with n ≥ 1, provided (T₂ ◦ T₁)(s) ̸= s. Now, we assume towards contradiction that for all 0 ≤ n ≤ 2 * | → | we always have (T₂ ◦ T₁)(s) ̸= (T₂ ◦ T₁)(s+1). We take n = m − 1 where m = 2 * | → |. Therefore, #((T₁ ◦ T₂)(s)) ≤ #(s) − m + 1. Since #(s) ≤ 2 * | → |, we conclude that #((T₂ ◦ T₁)(s)) ≤ 1. From Definition 4.13, we obtain that the set of transitions of reachable states of (T₂ ◦ T₁)(s) is either {(s, σ, s)} or {(s, τ, s′)}. Obviously none of the transformations T₁ or T₂ change the transition relation of (T₂ ◦ T₁)(s) in either of the above cases. Consequently, (T₂ ◦ T₁)(s) = (T₂ ◦ T₁)(s+1) is obtained for some 0 ≤ n ≤ 2 * | → |.

Suppose that (T₂ ◦ T₁)(s) = (T₂ ◦ T₁)(s+1) for some 0 ≤ n ≤ 2 * | → |. To prove the second item, we assume towards contradiction that init(s) ̸= {σ}; thus there exists a state s′ ∈ der(s) such that s′ ̸= s. Since (T₂ ◦ T₁)(s) is an SBN IOLTS\(^n\), we obtain that one of the transformations T₁ or T₂ is applicable on (T₂ ◦ T₁)(s). Subsequently, we find that (T₂ ◦ T₁)(s) ̸= (T₂ ◦ T₁)(s+1).

We can establish the theorem below which asserts that the testing power of IOLTS\(^n\)'s equals the power of testing of IOLTSs' over the set of IOTS\(^n\) input-enabled implementations.

**Theorem 4.19.** The testing power of IOLTS\(^n\) equals the testing power of IOLTS with respect to the ioco relation when implementations are in the intersection of IOTS and IOLTS\(^n\).

**Proof.** We need to show that

∀s ∈ IOLTS(I, U), ∃s′ ∈ IOLTS\(^n\)(I, U), ∀ť ∈ IOTS(I, U) ∩ IOTS\(^n\)(I, U) •

ť ioco s iff ř ioco s′

We take a specification s as an arbitrary IOLTS. In the remainder of the proof, we construct a testing-equivalent IOLTS\(^n\) s' for IOLTS s with respect to the set of IOTS\(^n\) input-enabled implementations. More formally, we build IOLTS\(^n\) s' such that the following statement always holds,

∀ť ∈ IOTS(I, U) ∩ IOTS\(^n\)(I, U) • ř ioco s iff ř ioco s'

To this end, the SBN IOLTS\(^n\) \langle Q, I, U ∪ {σ}, →, ř, q \rangle is constructed from the suspension automaton of IOLTS s according to the inference rules of Proposition 4.7. Following Proposition 4.18, there exists a finite number 1 ≤ n < 2 * | → | such that (T₂ ◦ T₁)(q) = (T₂ ◦ T₁)(q+1). Furthermore, (T₂ ◦ T₁)(q) has no σ-labeled transition at its reachable states from the initial state ř other than its initial state.

Now, we construct IOLTS\(^n\) \langle s', I, U, →, s', q' \rangle from (T₂ ◦ T₁)(q) as follows:
Chapter 4. Implementation Relations

- $S' = \text{der}(\bar{q})$
- $s' = \bar{q}$
- $\rightarrow_s = \begin{cases} \{(\bar{q}, x, \bar{q}) | \text{for some arbitrary } x \in U\} & \text{if } \bar{q} \xrightarrow{\sigma}_n \\ \{(s, a, s') | s, s' \in Q \land a \in I \cup U \cup \{\sigma\}\} & \text{otherwise} \end{cases}$

In the remainder of this proof, we show that IOLTS$\sqcap$ s has the same testing power as the IOLTS $s$. We distinguish two cases: $\bar{q} \nRightarrow_n$ and $\bar{q} \xrightarrow{\sigma}_n \bar{q}$.

- We suppose that $\bar{q} \nRightarrow_n$; thus $\rightarrow_s = \{(s, a, s') | s, s' \in \text{der}(q) \land a \in I \cup U \cup \{\sigma\}\}$. Following Proposition 4.18, it is obtained that none of the reachable states in $(T_2 \circ T_1)^n(\bar{q})$ has a $\sigma$-labeled transition; thus all labels of transitions in IOLTS$\sqcap$ $s'$ are in $L$, i.e., $\rightarrow_s \subseteq S' \times (I \cup U) \times S'$. Furthermore, since the set of transitions of reachable states in $s'$ is the same as the one in $(T_2 \circ T_1)^n(\bar{q})$, IOLTS$\sqcap$ $s'$ has the same testing power as $(T_2 \circ T_1)^n(\bar{q})$.

- We suppose that $\bar{q} \xrightarrow{\sigma}_n$. As mentioned before, transformations $T_1$ and $T_2$ maintain the structural properties defined in Definition 4.13; thus, IOLTS$\sqcap$ $(T_2 \circ T_1)^n(\bar{q})$ is an SBN IOLTS$\sqcap$. It is concluded from the last observation that $\bar{q} \xrightarrow{\sigma}_n \bar{q}$. Thus, the transition relation of IOLTS$\sqcap$ $s'$ would be $\{(s', x, s') | \text{for some arbitrary } x \in U\}$, as illustrated in the following picture.

Since the observation of quiescence at the initial state of every IOTS$\sqcap$ input-enabled implementation is possible, the set of conforming implementations of $(T_2 \circ T_1)^n(\bar{q})$ is empty. Using similar reasoning, IOLTS$\sqcap$ $s'$ does not recognize any IOTS$\sqcap$ input-enabled implementation as conforming, because $\delta \notin \text{out}(s' \text{ after } \epsilon)$. So, the testing powers of IOLTS$\sqcap$s $s'$ and IOLTS$\sqcap$ $(T_2 \circ T_1)^n(\bar{q})$ are the same.

It is directly concluded from Propositions 4.15 and 4.17 that the testing power of $(T_2 \circ T_1)^n(\bar{q})$ is the same as the testing power of IOLTS$\sqcap$ $\bar{q}$.

Combining the last two observations, we deduce that the set of conforming implementations of IOLTS$\sqcap$ $s'$ is the same as IOLTS$\sqcap$ $\bar{q}$. Consequently, Proposition 4.10 together with Proposition 4.12 yields that $\forall \tilde{r} \in \text{IOTS}(I, U) \cap \text{IOLTS}(I, U) \bullet \tilde{r} \text{ ioco } s \iff \tilde{r} \text{ ioco } s'$.  

4.3 The Effect of Observations on the Power of Testing

In this section, we investigate the effect of varying the set of observations $F$ on the testing power of the resulting conformance relations. Note that the question here is orthogonal
4.3. The Effect of Observations on the Power of Testing

to the one that we asked in the previous section: here we fix the specifications and ask whether by considering a subset of the set of observations \( F \), we obtain conformance relations that retain the testing power of the full set of observations \( F \). The proposition below states that the testing power of \( \text{ioco}_F^{a,b} \) is monotonic in the set of observations \( F \); from this, it follows that testing power may be affected by considering different sets \( F \).

**Proposition 4.20.** Let \( F, F' \subseteq L^*_\delta \). Then \( F' \subseteq F \) implies \( \text{ioco}_F^{a,b} \subseteq \text{ioco}_{F'}^{a,b} \).

**Proof.** An immediate consequence of Definition 4.1. \( \square \)

We are, in particular, interested in suspension traces that naturally capture the observations that we can make of an IOTS\( ^\wedge \) implementation. The crucial difference between IOTS\( ^\wedge \) implementations and IOTS implementations is that the latter are always willing to accept inputs, whereas the former only accepts inputs when we can also observe quiescence. For IOTS\( ^\wedge \) implementations, providing inputs in any other situation is undesirable, and, hence, reasoning about traces that would attempt to do so in our conformance relation would be equally undesirable. We therefore introduce a new class of traces, called *internal-choice traces*, which naturally characterize the observable behaviors of IOTS\( ^\wedge \) implementations.

**Definition 4.21 (Internal-choice traces).** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOLTS. Let \( s \in S \) be an arbitrary state and \( \sigma \in L^*_\delta \). The set of *internal-choice traces* of \( s \), denoted by \( \text{ICtraces}(s) \) is a subset of suspension traces in which quiescence is observed before every input action, i.e., \( \text{ICtraces}(s) = \text{Straces}(s) \cap (U \cup (\{\delta\}^+ I) \cup \{\delta\})^* \); we set \( \text{ICtraces}(S') = \bigcup_{s \in S'} \text{ICtraces}(s) \) for \( S' \subseteq S \).

Note that, as a result of Proposition 4.20, using internal-choice traces instead of suspension traces leads to a weaker testing relation. It is not, however, immediate that the inclusion of Proposition 4.20 is strict. Table 4.2 summarizes our results in this sections. In each row the testing power of IOLTS\( ^\wedge \) and IOTS are compared in testing implementations that behave as the mentioned subclass of IOLTS.

Table 4.2: Comparison of the testing power of suspension traces vs. internal-choice traces

<table>
<thead>
<tr>
<th>Model of Specification ( s )</th>
<th>Model of implementation</th>
<th>Testing power of ( \text{Straces}(s) ) vs. ( \text{ICtraces}(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOLTS</td>
<td>IOTS</td>
<td>Different</td>
</tr>
<tr>
<td>IOLTS( ^\wedge )</td>
<td>IOTS</td>
<td>Different</td>
</tr>
<tr>
<td>IOLTS( ^\wedge )</td>
<td>IOTS( ^\wedge )</td>
<td>Different</td>
</tr>
<tr>
<td>IOLTS( ^\wedge )</td>
<td>IOTS( ^\wedge )</td>
<td>Same</td>
</tr>
</tbody>
</table>

The following example shows that restriction the set of observation to internal-choice traces in the standard *ioco* testing leads to a weaker implementation relation.

69
Chapter 4. Implementation Relations

Figure 4.7: Two different specifications (a) and (b) of a vending machine and an implementation (c) illustrating that the testing power of internal-choice traces is strictly less than the testing power of suspension traces in the family of conformance relations \( \text{ioco}_F \).

Example 4.22. Let \( c_0 \) be the specification depicted in Figure 4.7(a) and let \( \tilde{i} \) in Figure 4.7(c) be its implementation. Following Definition 4.1, \( \tilde{i} \text{ioco} c_0 \), because the observed output coffee in the implementation after the execution of trace \( \text{coin \ button} \) is not allowed by specification \( c_0 \) after that trace. The set \( \text{ICtraces}(c_0) = \{ \epsilon, \delta \sigma \text{coin, } \delta \sigma \text{coin refund} \mid \sigma \in \delta^* \} \). Clearly, for all \( \sigma \in \text{ICtraces}(c_0) \), we have \( \text{out}(\tilde{i} \text{ after } \sigma) \subseteq \text{out}(c_0 \text{ after } \sigma) \). Hence, \( \tilde{i} \text{ioco} \text{ICtraces}(c_0) c_0 \).

Theorem 4.23. Let \( s \in \text{IOLTS}(I, U) \) be a specification. Then,

\[
\text{ioco}_{\text{Straces}(s)} \subset \text{ioco}_{\text{ICtraces}(s)}
\]

Proof. It is a direct consequence of Definition 4.20 together with example 4.22. \( \square \)

We next consider restricting the set of observations \( F \) to internal-choice traces in the conformance family \( \text{ioco}^\text{I} \) and compare the resulting testing power to the one obtained using suspension traces. As illustrated by the example below, restricting the set of specifications to internal-choice labeled transition systems is not a sufficient condition to retain the testing power of the full set of suspension traces.

Example 4.24. Consider again Figure 4.7. Take \( \text{IOLTS}^\text{I} o_0 \) as specification and again consider \( \tilde{i} \) as its implementation. Clearly, we have \( \tilde{i} \text{ioco}^\text{I} \text{Straces}(o_0) o_0 \). For instance, considering trace \( \text{coin \ button} \), we find that \( \text{out}(\tilde{i} \text{ after } \text{coin button}) = \{\text{coffee}\} \), whereas it is not
allowed in the outputs of $o_0$ after that trace, i.e., $\text{out}(o_0 \text{ after coin button}) = \{\text{tea}\}$. In conformance testing with respect to $\text{io}_\text{Ctraces}(o_0)$, trace $\delta_\text{coin} \delta_\text{button}$ is examined instead of trace $\text{coin button}$. We find that $\text{out}(i \text{ after } \delta_\text{coin} \delta_\text{button}) = \text{out}(o_0 \text{ after } \delta_\text{coin} \delta_\text{button}) = \emptyset$. It is obtained by checking all other traces in $\text{ICtraces}(o_0)$ that $i \text{io}_\text{Ctraces}(o_0) o_0$.

**Theorem 4.25.** Let $s \in \text{IOLTS}_F(I, U)$ be a specification. Then,

$$\text{io}_\text{Straces}(s) \subset \text{io}_\text{ICtraces}(s)$$

**Proof.** This proof is a direct consequence of Definition 4.20 together with example 4.24.

We next investigate whether switching to a different model of implementations will change these results: we henceforth assume that implementations can be modeled using IOTS$^\cap$s. The example below shows that, assuming that specifications can still be arbitrary IOLTS$s$, the testing power of using internal-choice traces is inferior to using suspension traces.

![Figure 4.8: A specification (a) and an implementation (b) illustrating that the testing power of internal-choice traces is strictly less than the testing power of suspension traces in the family of conformance relations $\text{io}_F^n$.

**Example 4.26.** Consider IOLTS $\bar{s}$ in Figure 4.8. Analogous to the IOLTS$s$ in Figure 4.7, it models a vending machine which after receiving money, either refunds or accepts it; if accepted, coffee is produced after pressing a coffee button (in this case, $\text{button}_c$), and, similarly, tea is produced after pressing a tea button ($\text{button}_t$). The transition system $\bar{i}$ is input-enabled only at quiescent states, i.e., it is an IOTS$^\cap$. Take IOTS$^\cap$ $\bar{i}$, also in Figure 4.8, as a potential implementation. Regarding Definition 4.1, we find that $\bar{i} \text{io}_F^n \bar{s}$, because specification $\bar{s}$ after executing trace $\text{coin button}$ allows only output coffee, whereas $\bar{i}$ after the same trace produces tea. Obviously, we have $\text{out}(\bar{i} \text{ after } \sigma) \subset \text{out}(\bar{s} \text{ after } \sigma)$ for every $\sigma \in \text{ICtraces}(\bar{s})$, where $\text{ICtraces}(\bar{s}) = \{\varepsilon, \sigma \delta_\text{coin}, \sigma \delta_\text{coin} \delta_\text{refund}, \sigma \delta_\text{coin} \delta_\text{button}_t, \sigma \delta_\text{coin} \delta_\text{button}_t \text{ tea} | \sigma \in \delta^*\}$. We therefore obtain that $\bar{i} \text{io}_F^n \bar{s}$ with $F = \text{ICtraces}(\bar{s})$.

**Theorem 4.27.** Let $s \in \text{IOLTS}(I, U)$ be a specification. Then,

$$\text{io}_\text{Straces}(s) \subset \text{io}_\text{ICtraces}(s)$$
Chapter 4. Implementation Relations

Proof. This proof is a direct consequence of Definition 4.20 together with example 4.26.

Finally, we investigate the case that specifications are assumed to be internal-choice IOLTSs. The result below shows that, contrary to the previous cases we analyzed, the resulting conformance relations for internal-choice traces and suspension traces coincide.

Theorem 4.28. Let \( s \in IOLTS^\Pi(I, U) \) be a specification and \( \bar{i} \in IOTS^\Pi(I, U) \) be an implementation. Then,

\[
\bar{i} \text{ ioco}_{ICtraces(s)}^\Pi s \text{ iff } \bar{i} \text{ ioco}_{Straces(s)}^\Pi s
\]

Proof. The implication from right to left is an instance of Proposition 4.20. We therefore focus on the implication from left to right.

We first show that for every \( \sigma \in Straces(s) \), there is some \( \sigma' \in ICtraces(s) \) such that both \( s \) after \( \sigma = s \) after \( \sigma' \) and \( \bar{i} \) after \( \sigma = \bar{i} \) after \( \sigma' \). We do this by induction on the number of input actions in \( \sigma \).

- **Base case.** For the induction basis assume that \( \sigma \in (U \cup \{\delta\})^* \). Following Definition 4.21, \( \sigma \in ICtraces(s) \). Hence, \( \sigma' = \sigma \) satisfies the required condition.

- **Induction step.** Assume for the induction step that the given claim holds for all sequences with \( n - 1 \) input actions. Suppose that we have a sequence \( \sigma \) with \( n \) input actions; that is, \( \sigma = \sigma_1 a \sigma_2 \) with \( \sigma_1, \sigma_2 \in (U \cup \{\delta\})^* \) and \( a \in I \). Thus, \( \sigma_1 \) has \( n - 1 \) input actions.

Following the induction hypothesis, there exists a \( \sigma'_1 \in ICtraces(s) \) such that we have \( s \) after \( \sigma_1 = s \) after \( \sigma'_1 \) and \( \bar{i} \) after \( \sigma_1 = \bar{i} \) after \( \sigma'_1 \) hold. We conclude from \( s \in IOLTS^\Pi(I, U) \) along with \( \sigma_1 a \in Straces(s) \) that there exists a non-empty subset of states in \( s \) after \( \sigma_1 \) consisting of quiescent states. Suppose \( S' \) is the largest possible set of quiescent states in \( s \) after \( \sigma_1 \). By definition of IOLTSs (see Definition 2.15), we that \( s \) after \( \sigma_1 a \sigma_2 = S' \) after \( a \sigma_2 \). Consequently, by substituting \( S' \) with \( s \) after \( \sigma'_1 \delta \) we have \( s \) after \( \sigma = s \) after \( \sigma'_1 \delta a \sigma_2 \).

It follows from Definition 4.21 that \( \sigma'_1 \delta a \sigma_2 \in ICtraces(s) \). Therefore, \( s \) after \( \sigma = s \) after \( \sigma'_1 \delta a \sigma_2 \) holds. Along the same lines of reasoning, we can show that for the same internal-choice trace we have \( \bar{i} \) after \( \sigma = \bar{i} \) after \( \sigma'_1 \delta a \sigma_2 \).

We next prove the property by contraposition. Suppose that \( \bar{i} \text{ ioco}_{Straces(s)}^\Pi s \). Then for some \( \sigma \in Straces(s) \), \( \text{out}(\bar{i} \text{ after } \sigma) \not\subseteq \text{out}(s \text{ after } \sigma) \). By the above result, we find that there must be some \( \sigma' \in ICtraces(s) \) such that \( \bar{i} \text{ after } \sigma = \bar{i} \text{ after } \sigma' \) and \( s \text{ after } \sigma = s \text{ after } \sigma' \). But then also \( \text{out}(\bar{i} \text{ after } \sigma') \not\subseteq \text{out}(s \text{ after } \sigma') \). So, it also must hold that \( \bar{i} \text{ ioco}_{ICtraces(s)}^\Pi s \).

As an immediate consequence of Theorem 4.28, for implementations in the intersection of IOTS\(^\Pi\) and IOTS, the testing power of \( \text{ioco}_{ICtraces(s)}^\Pi \) and that of the standard \( \text{ioco} \) coincide, as stated by the proposition below.

Proposition 4.29. Let \( s \in IOLTS^\Pi(I, U) \) be a specification and \( \tilde{r} \in IOTS^\Pi(I, U) \cap IOTS(I, U) \) be an implementation. Then \( \tilde{r} \text{ ioco}_{ICtraces(s)}^\Pi s \) iff \( \tilde{r} \text{ ioco} s \).

72
4.4 Test Case Generation

The definition of the family of conformance relations introduced and studied in the previous section assumes that we can reason about implementations as if these were transition systems we can inspect. Since in practice, this is often not the case (we only know that a model exists that underlies such an implementation), the definition cannot be used to check whether an implementation conforms to a given specification.

This problem can be sidestepped if there is a set of test cases that can be run against an actual implementation, and which have exactly the same discriminating power as the specification. In this section, we study the test cases that are needed to test for the family of conformance relations introduced in the previous section.

Generally, a test case prescribes when to stimulate an IUT by sending an input, and when to observe the outputs emitted by the IUT. Therefore, the inputs and the outputs are interchanged from the viewpoint of the tester. Henceforth, we refer to the inputs and the outputs from the viewpoint of the IUT. As mentioned in Section 2.2, the behavioral model of a test case is described by an IOLTS which is acyclic in order to guarantee the termination of the test execution of the test case against the IUT. Moreover, to formally distinguish the quiescence of the IUT, \( \theta \) action is added to the set of inputs of test cases which synchronizes with the output action \( \delta \) in the IUT.

An execution of a test case against the IUT is defined as the communication between them as formalized in Section 2.2. The testing theories studied in the previous section assume that the test execution is synchronously carried out, meaning that the tester and the IUT synchronize on their inputs and outputs. For the formal notation of test execution and also the definition of test verdicts, we refer to Section 2.2.

Each instance of the implementation relation of the \( \text{ioco}^{p,b}_a \) family, inspects a part of the behavior of the IUT. The set of behaviors explored is influenced by the parameter \( F \). Therefore, for every conformance relation \( \text{ioco}^{a,b}_p \), we need a set of test cases that capture the whole set of traces denoted by \( F \). In order to examine the set of internal-choice traces, in [Wei09; WW09] a subclass of test cases is introduced; test cases in this subclass are called internal-choice test cases, given in Definition 2.33 in Chapter 2. Such test cases stimulate an IUT only when quiescence has been observed. Intuitively, this will ensure that the test case is actually executable for implementations that behave as an IOTS. The property below provides us with an alternative characterization of an internal-choice test case.

**Property 4.30.** A test case \( t \) is an internal-choice test case, i.e., \( t \in \text{TTS}^\text{in}(U \cup \{\theta\}, I) \) iff \( \text{traces}(t) \subseteq (U \cup \{\{\theta\}^+I \cup \{\theta\})^* \).

**Example 4.31.** IOLTSs \( t \) and \( t' \) in Figure 4.9 show two test cases for IOTS \( o_0 \) in Figure 4.7. IOLTS \( t \) is an internal-choice test case. In this test case, inputs for the implementation are enabled only in states reached by a \( \theta \)-transition.

We next introduce a test case generation algorithm, based on Tretmans’ original algorithm [Tre96c], that is suited for testing against a conformance relation \( \text{ioco}^{a,b}_p \). The set of test cases generated by this algorithm is both sound and exhaustive with respect to a given implementation relation \( \text{ioco}^{a,b}_p \) for a given set of traces \( F \).

Note that Tretmans’ original test case generation algorithm [Tre96c] did not produce test cases that were input enabled. However, this issue was addressed several years ago.
Figure 4.9: Two test cases for IOLTS $\theta_0$ in Figure 4.7 with the set of inputs \{coin, button\} and the set of outputs \{tea, refund\}

in [Tre08], in which the algorithm for (plain) ioco was made to generate test cases that, in all non-terminal states, are willing to accept all the outputs produced by an implementation. We have used the ideas of the latter algorithm and incorporated them in Tretmans’ original algorithm.

Algorithm 4.4.1: TestCaseGeneration($S', F$)

\textbf{comment:} Let IOLTS $\langle S, I, U, \rightarrow, \hat{s} \rangle$ be a specification. Let $S' \subseteq S$, and let $F \subseteq \text{Straces}(S')$; then a test case $t \in \text{TTS}(U \cup \{\theta\}, I)$ is obtained by finite recursive applications of one of the following nondeterministic choices:

\begin{align*}
\text{t} & := \text{pass} \\
\text{t} & := \Sigma\{\hat{x}; \text{fail} \mid x \in U, x \not\in \text{out}(S'), e \in F\} \\
& \quad \Box \Sigma\{\hat{x}; \text{pass} \mid x \in U, x \not\in \text{out}(S'), e \not\in F\} \\
& \quad \Sigma\{\hat{x}; t_x \mid x \in U, x \in \text{out}(S')\},
\end{align*}

where $t_x$ is obtained by recursively applying the algorithm for \{\sigma \in L_0^+ \mid x\sigma \in F\} and $S'$ after $x$

\begin{align*}
& \quad \Box a; t_a, \\
& \quad \text{where } a \in I, \text{ such that } F' = \{\sigma \in L_0^+ \mid a\sigma \in F\} \neq \emptyset \text{ and } t_a \text{ is obtained by recursively applying the algorithm for } F' \text{ and } S' \text{ after } a \\
\text{t} & := \Sigma\{\hat{x}; \text{fail} \mid x \in U \cup \{\delta\}, x \not\in \text{out}(S'), e \in F\} \\
& \quad \Box \Sigma\{\hat{x}; \text{pass} \mid x \in U \cup \{\delta\}, x \not\in \text{out}(S'), e \not\in F\} \\
& \quad \Sigma\{\hat{x}; t_x \mid x \in U \cup \{\delta\}, x \in \text{out}(S')\},
\end{align*}

where $t_x$ is obtained by recursively applying the algorithm for \{\sigma \in L_0^+ \mid x\sigma \in F\} and $S'$ after $x$

In order to concisely describe Algorithm 4.4.1, we borrow Tretmans’ notation (see for instance [Tre08]) for behavioral expressions using the operators $;$, $\Box$ and $\Sigma$. Such behav-
ioral expressions represent transition systems. Informally, for an action label \(a\) (taken from some set of actions), and a behavioral expression \(B\), the behavioral expression \(a;B\) denotes the transition system that starts with executing the \(a\) action, leading to a state that behaves as \(B\). For a countable set of behavioral expressions \(\mathcal{B}\), the choice expression \(\Sigma \mathcal{B}\) denotes the transition system that, from its initial state, can nondeterministically choose between all behaviors described by the expressions in \(\mathcal{B}\). Expression \(B_1 \sqcup B_2\), for behavioral expressions \(B_1\) and \(B_2\), is used as an abbreviation for \(\Sigma \{B_1, B_2\}\), i.e., it behaves either as \(B_1\) or \(B_2\).

Upon termination, algorithm 4.4.1 generates a test case for a set of states \(S'\) and a subset of its suspension traces \(F\) of a given specification \(\bar{s} \in \text{IOLTS}((I, U))\). The parameters \(S'\) is typically initialized as \(\bar{s}\) after \(\epsilon\).

The proposition below establishes a formal connection between a subset of the suspension traces of a given specification, and the traces of the test cases generated with Algorithm 4.4.1 for that specification. The proposition, which is similar to Tretmans’ Lemma A.25 in [Tre96c], is essential in establishing the exhaustiveness of the test case generation algorithm.

**Proposition 4.32.** Let \(\langle S, I, U, \rightarrow, \bar{s} \rangle\) be an IOLTS. Let \(F \subseteq \text{Straces}(S')\) with \(S' \subseteq S\), let \(\sigma \in F\). Define \(t_{[\sigma,F,S']}\) by:

\[
\begin{align*}
t_{[\epsilon,F,S']} & = \Sigma\{\bar{x}; \text{fail} \mid x \in U \cup \{\delta\}, x \notin \text{out}(S')\} \\
\Delta & \Sigma\{\bar{x}; \text{pass} \mid x \in U \cup \{\delta\}, x \in \text{out}(S')\}
\end{align*}
\]

\[
t_{[a;\sigma,F,S']} \quad (\bar{a} \in I) = \Sigma\{\bar{x}; \text{fail} \mid x \in U, x \notin \text{out}(S'), e \in F\} \\
\Delta \Sigma\{\bar{x}; \text{pass} \mid x \in U, x \notin \text{out}(S'), e \notin F\} \\
\Delta \Sigma\{\bar{x}; \text{pass} \mid x \in U, x \in \text{out}(S')\} \\
\Delta \bar{a}; t_{[\sigma,F',S']} \\
\quad \text{(where } F' = \{\sigma' \in L_0^\epsilon \mid a\sigma' \in F\} \text{ and } S'' = S' \text{ after } a\)
\]

\[
t_{[y;\sigma,F,S']} \quad (\bar{y} \in U \cup \{\delta\}) = \Sigma\{\bar{x}; \text{fail} \mid x \in U \cup \{\delta\}, x \notin \text{out}(S'), e \in F\} \\
\Delta \Sigma\{\bar{x}; \text{pass} \mid x \in U \cup \{\delta\}, x \notin \text{out}(S'), e \notin F\} \\
\Delta \Sigma\{\bar{x}; \text{pass} \mid x \in U \cup \{\delta\}, x \in \text{out}(S'), x \neq y\} \\
\Delta \bar{y}; t_{[\sigma,F',S']} \\
\quad \text{(where } F' = \{\sigma' \in L_0^\epsilon \mid y\sigma' \in F\} \text{ and } S'' = S' \text{ after } y\)
\]

then

1. \(t_{[\sigma,F,S']}\) can be obtained from \(F\) and \(S'\) with Algorithm 4.4.1

2. \(x \notin \text{out}(S' \text{ after } \sigma)\) implies \(t_{[\sigma,F,S']} \quad \xrightarrow{\sigma x} \text{fail}\)

**Proof.** The proof is straightforward by considering the structure of \(t_{[\sigma,F,S']}\) and using an induction on \(\sigma\); the proof is very similar to the proof in [Tre96c, Lemma A.25].

**Theorem 4.33.** Let IOLTS \(\langle S, I, U, \rightarrow, \bar{s} \rangle\) be a specification. Then

1. a test case obtained with Algorithm 4.4.1 from \(\bar{s}\) after \(\epsilon\) and \(F \subseteq \text{Straces}(\bar{s})\) is sound for \(\bar{s}\) with respect to \(\text{ioco}_{F}^{a,b}\) for \(a, b \in \{\land, \lor\}\).

2. the set of all possible test cases that can be obtained by Algorithm 4.4.1 from \(\bar{s}\) after \(\epsilon\) and \(F \subseteq \text{Straces}(\bar{s})\) is exhaustive for \(\bar{s}\) with respect to \(\text{ioco}_{F}^{a,b}\) for \(a, b \in \{\land, \lor\}\).
Proof. The soundness of test cases generated by Algorithm 4.4.1 is proved by induction on the structure of them; the exhaustiveness of the algorithm follows from Proposition 4.32. The proof is similar to the proof of Theorem 6.3 in [Tre96c]; □

Observe that the above theorem does not imply that the test cases derived by Algorithm 4.4.1 can be executed successfully on both classes of implementations that we discussed in the previous sections. Whereas for Tretmans’ implementations behaving as IOTSs, successful test case execution is no issue, this is not the case for Weiglhofer and Wotawa’s implementations behaving as IOTS$^{\cap}$s. For the latter class of implementations it is possible that the test case is forced to observe outputs, since the implementation is unwilling to accept stimuli from the test case. Thus, it makes no sense to consider such test cases, as the example below illustrates.

Example 4.34. Consider again Figure 4.9. Take IOLTS $t’$ as the test case generated with Algorithm 4.4.1 from IOTS $o_0$ and sequence coin button and take IOTS$^{\cap} d_0$, depicted in Figure 4.10 as a potential implementation. Consider the execution $t_0^’ | d_0 \,吃完 \, t_1^’ | d_1$ where $t_0^’$ is the initial state of IOLTS $t’$ and $t_1^’ = t_0^’ \, after \, coin$. At state $t_1^’$, test case $t’$ can try to provide the input button to the IUT while IOTS$^{\cap} d_0$ is not willing to accept any inputs. Therefore, the test case is prevented from executing the sequence coin button.

To cope with the issue of successful executability of test cases, we next investigate when our test case generation algorithm can be made to produce only executable test cases, while still guaranteeing soundness and exhaustiveness. Our studies of the ioco family of conformance relations in the previous section are essential in establishing the latter results.

First, we have the following technical lemma and proposition which state that traces of a test case can be chopped up into smaller traces.

Lemma 4.35. Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOLTS. Let $S’ \subseteq S$ be a set of states and $F \subseteq \text{Straces}(S’)$. Then for all $y \sigma \in F$ we have:

$$\text{traces}(t_{[y \sigma, FS']}) = \{\epsilon\} \cup U \cup \{\theta \mid y \notin I\} \cup \{y\} \cup \{\overline{\rho} \mid \rho \in \text{traces}(t_{[\sigma, (y \rho \in F), S’ after y]})\}$$

Proof. Follows immediately from the definition of traces($t_{\sigma, FS’}$). □

Proposition 4.36. Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOLTS. Let $S’ \subseteq S$ be a set of states and $F \subseteq \text{Straces}(S’)$. Then for all $\sigma_1 \sigma_2 \in F$ satisfying $\sigma_1 \neq \epsilon$, we have:

$$\text{traces}(t_{[\sigma_1 \sigma_2, FS’]}) = \text{traces}(t_{[\sigma_1, FS’]} \cup \{\overline{\sigma_1 \rho} \mid \rho \in \text{traces}(t_{[\sigma_2, (\rho \sigma_1 \rho \in F), S’ after \sigma_1])\})$$

Proof. The proof proceeds by induction on the length of $\sigma_1$.

• Base case. Follows immediately from Lemma 4.35.

• Induction step. Assume for the induction step that the above statement holds for all sequences of length $n – 1$ and the length of $\sigma_1$ is $n$. Suppose $\sigma_1 = x \sigma_1’$ with $\sigma_1’ \in L_\sigma^*$ and $x \in L_\delta$. Therefore, the length of $\sigma_1’$ is $n – 1$. 

76
Because of Property 4.30, it suffices to show that

\[ \text{Proof.} \]

Proposition 4.38. relies on Property 4.30. cases only. The propositions below confirm that this is indeed possible. This proposition whether the test case generation algorithm can be made to generate internal-choice test case and an IOLTS proceeds in an orchestrated fashion: the IOLTS is only consuming the stimulus.

Together, † and ‡ yield the desired equivalence.

\[ \square \]

The proposition given below formalizes that, indeed, the interaction between an internal-choice test case and an IOLTS proceeds in an orchestrated fashion: the IOLTS is only provided a stimulus whenever it has reached a stable situation, and is thus capable of consuming the stimulus.

**Proposition 4.37.** Let \( s \) be an arbitrary IOLTS and \( t \) be an internal-choice test case. Let \( x \in I \). Then for all \( \sigma \in L_\theta^* \), we have:

\[ t \upharpoonright s \overset{\sigma^*}{\rightarrow} \] implies \( \exists \sigma' \in L^*, \sigma_{\theta} \in \theta^* : \sigma = \sigma' \theta \sigma_{\theta} \]

*Proof.* It is a direct consequence of Property 4.30.

Due to the above propositions, we can thus guarantee that test cases are successfully executable on implementations that behave as IOTS’s. It thus suffices to investigate whether the test case generation algorithm can be made to generate internal-choice test cases only. The propositions below confirm that this is indeed possible. This proposition relies on Property 4.30.

**Proposition 4.38.** Let \( \langle S, I, U, \rightarrow, s \rangle \) be an IOLTS. Then for all \( S' \subseteq S \), all \( F \subseteq \text{ICtraces}(S') \) and all \( \sigma \in F \), the test case \( t_{[\sigma,F,S']} \) is an internal-choice test case.

*Proof.* Because of Property 4.30, it suffices to show that \( \text{traces}(t_{[\sigma,F,S']}) \subseteq (U \cup (\{\theta\}^* U) \cup \{\theta\})^* \). We prove it by induction on the number of input actions in \( \sigma \).
Chapter 4. Implementation Relations

- **Base case.** Assume for the basis of the induction that \( \sigma \in (U \cup \{ \delta \})^* \). We proceed by a second induction on the length of \( \sigma \).

  - **Base case.** Suppose \( \sigma = \epsilon \). We find from Proposition 4.32 that \( \text{traces}(t_{\epsilon,F,S'}) = \{ \epsilon \} \cup U \cup \{ \theta \} \): \( t_{\epsilon,F,S'} \) has an \( x \)-labeled transition to the pass state for \( x \in \text{out}(\Sigma') \), and to the fail state for \( x \not\in \text{out}(\Sigma') \). Clearly, \( U \cup \{ \theta \} \in (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \). Hence, \( t_{\epsilon,F,S'} \) is an internal-choice test case.

  - **Induction step.** Assume for the induction step of the second induction that the above statement holds for all sequences of length \( n - 1 \) and that the length of \( \sigma \) is \( n \). Take \( \sigma = y\sigma' \) with \( y \in U \cup \{ \delta \} \). Following Lemma 4.35, \( \text{traces}(t_{y\sigma',F,S'}) = \{ \epsilon \} \cup (U \cup \{ \theta \}) \cup \{ \bar{y}\rho \mid \rho \in \text{traces}(t_{\sigma',F,S'}) \} \) with \( \rho' = \{ \rho' \mid \sigma_1\rho' \in F \} \) and \( S'' = S' \) after \( y \). We know from our induction hypothesis that \( \text{traces}(t_{\sigma',F,S''}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \). Consequently, we find that \( \{ \bar{y}\rho \mid \rho \in \text{traces}(t_{\sigma',F,S'}) \} \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \). Combined with our previous observations, we find that \( \text{traces}(t_{\sigma,F,S'}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \).

- **Induction step.** Assume for the induction step that our statement holds for all sequences with \( n - 1 \) input actions. Let \( \sigma \in F \) be a sequence containing \( n \) input actions, where \( F \subseteq \text{ICtraces}(\Sigma') \); assume \( \sigma = \sigma_1 \delta a \sigma_2 \), where \( \sigma_1 \in L_a^* \), \( a \in I \) and \( \sigma_2 \in U^* \). We distinguish two cases; \( \sigma_1 = \epsilon \), and \( \sigma_1 \neq \epsilon \). First, we assume that \( \sigma_1 = \epsilon \). We therefore need to show that \( \text{traces}(t_{\delta a\sigma_2,F,S'}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \) holds.

We next suppose that \( \sigma_1 \neq \epsilon \). From Proposition 4.36, we find that \( \text{traces}(t_{\sigma,F,S'}) = \text{traces}(t_{\delta a,F,S'}) \cup \{ \overline{\delta a}\rho \mid \rho \in \text{traces}(t_{\sigma_2,F,S''}) \} \) where \( F' = \{ \rho' \mid \sigma_1\rho' \in F \} \) and \( S'' = S' \) after \( \sigma_1 \). From our induction hypothesis, we know that \( \text{traces}(t_{\sigma_1,F,S''}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \). It thus suffices to show that \( \{ \overline{\delta a}\rho \mid \rho \in \text{traces}(t_{\sigma_2,F,S''}) \} \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \). We know from \( \sigma_1 \in \text{ICtraces}(\Sigma') \) that \( \overline{\sigma_1} \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \). Therefore, \( \{ \overline{\delta a}\rho \mid \rho \in \text{traces}(t_{\sigma_2,F,S''}) \} \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \) follows if we can prove that \( \text{traces}(t_{\delta a\sigma_2,F,S''}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \) holds.

Therefore, in the two above cases it suffices to show that \( \text{traces}(t_{\delta a\sigma_2,F,S''}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \) holds.

By applying Proposition 4.36 twice, we find that

\[
\text{traces}(t_{\delta a\sigma_2,F,S''}) = \text{traces}(t_{\delta a,F,S''}) \cup \text{traces}(t_{\sigma_2,F,S''}) \cup \{ \overline{\delta a}\rho \mid \rho \in \text{traces}(t_{\sigma_2,F,S''}) \}
\]

with \( F'_a = \{ \rho' \mid \delta a\rho' \in F' \} \), \( F'' = \{ \rho'' \mid \delta a\rho'' \in F' \} \), \( S'' = S'' \) after \( \delta \) and \( S'' = S'' \) after \( a \).

From Lemma 4.35, \( \text{traces}(t_{\delta a,F,S''}) = \{ \epsilon \} \cup U \cup \{ \theta \} \) and also \( \text{traces}(t_{\sigma_2,F,S''}) = \{ \epsilon \} \cup U \cup \{ \theta \} \) are obtained. Following the base case of our induction, we find that \( \text{traces}(t_{\delta a\sigma_2,F,S''}) \subseteq (U \cup \{ \theta \})^* \) as well. Combining all observations, we find that

\[
\text{traces}(t_{\delta a\sigma_2,F,S''}) = \{ \epsilon \} \cup U \cup \{ \theta \} \cup \{ \overline{\delta a}x \mid x \in U \} \cup \{ \overline{\delta a}\rho \mid \rho \in (U \cup \{ \theta \})^* \}
\]

From this, we obtain \( \text{traces}(t_{\delta a\sigma_2,F,S''}) \subseteq (U \cup (\{ \theta \}^+I) \cup \{ \theta \})^* \), which was to be shown.
The proposition below formalizes the set of traces of test cases generated with Algorithm 4.4.1 for given subsets of suspension traces and states of a given specification. The proposition is essential to ensure that for a given subset of internal-choice traces Algorithm 4.4.1 generates test cases that are always executable against implementations behaving as IOTs’.

**Proposition 4.39.** Let \( \langle S, I, U, \rightarrow, \tilde{s} \rangle \) be an IOLTS, let \( F \subseteq \text{straces}(S') \) with \( S' \subseteq S \), and let \( T \) be a set of test cases obtained with Algorithm 4.4.1 from \( S' \) and \( F \). We have \( \text{traces}(T) \subseteq \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \).

**Proof.** The proof is given by induction on the number of recursions of Algorithm 4.4.1 in generating a test case \( t \in T \).

- **Base case.** We assume for the induction basis that test case \( t \) is generated by one recursive application of the algorithm. It is obvious that \( t := \text{pass} \). It follows from \( \text{traces(pass)} = \varepsilon \) that \( \text{traces(pass)} \subseteq \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \).

- **Induction step.** For the induction basis assume that the above thesis holds for all test cases obtained from \( n \) times or less recursive application of the algorithm and test case \( t \) is generated from \( n \) times recursion. We distinguish two cases.

  - We suppose the second choice of the algorithm is selected at the first round of the application of the algorithm. Following the algorithm, \( \text{traces}(t) = \{ \tilde{x} \mid x \notin \text{out}(S) \} \cup \bigcup_{x \notin \text{out}(S)} \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_x) \} \cup \{ \tilde{a}\rho \mid a \in I, \rho \in \text{traces}(t_a) \} \).

    * We consider \( x \notin \text{out}(S) \). Upon observing \( x \notin \text{out}(S) \), \( t \) goes to terminal states and the algorithm terminates. Therefore, \( t \) is obtained by applying the algorithm once. Following the induction hypothesis, \( \{ \tilde{x} \mid x \notin \text{out}(S) \} \subseteq \bigcup_{\sigma \in F} t_{[\sigma,F,S]} \).

    * We suppose that \( t := x_t \) for some \( x \in \text{out}(S) \). We know that \( t_x \) is obtained by recursively applying the algorithm for \( F' = \{ \sigma \mid x\sigma \in F \} \) and \( S' = S \) after \( x \). Clearly, \( t_x \) is obtained by at most \( n-1 \) applications of the algorithm. It follows from the induction hypothesis that \( \text{traces}(t_x) \subseteq \bigcup_{\sigma \in F'} \text{traces}(t_{[\sigma,F',S']}) \). We know from Lemma 4.35 that for every \( \sigma \in F' \), \( \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_{[\sigma,F',S']}) \} \subseteq \{ \text{traces}(t_{[x\sigma,F,S]}) \} \) (note that for all sequence \( x\sigma \in F \), we know that \( x\sigma \in F \)). Therefore, the previous observation along with \( \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_x) \} \subseteq \bigcup_{\sigma \in F'} \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_{[\sigma,F',S']}) \} \) leads to \( \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_x) \} \subseteq \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \). Consequently, \( \bigcup_{x \in \text{out}(S)} \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_x) \} \subseteq \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \) follows.

    * We suppose that \( t := a_t \) for some \( a \in I \) where \( F' = \{ \sigma \mid a\sigma \in F \} \neq \emptyset \) and \( t_a \) is obtained recursively by applying the algorithm for \( F' \) and \( S' = S \) after \( a \). With the same lines of reasoning in the previous item, we conclude that \( \{ \tilde{a}\rho \mid \rho \in \text{traces}(t_a) \} \subseteq \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \).

Therefore, we showed that all three sets \( \{ \tilde{x} \mid x \notin \text{out}(S) \} \), \( \bigcup_{x \notin \text{out}(S)} \{ \tilde{x}\rho \mid \rho \in \text{traces}(t_x) \} \), and \( \{ \tilde{a}\rho \mid a \in I, \rho \in \text{traces}(t_a) \} \) are a subset of \( \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \). Hence, \( \text{traces}(t) \subseteq \bigcup_{\sigma \in F} \text{traces}(t_{[\sigma,F,S]}) \).
– We suppose the third choice of the algorithm is selected at the first round of the application of the algorithm. Following the algorithm, \( \text{traces}(t) = \{ \bar{x} \mid x \not\in \text{out}(S) \} \cup \bigcup_{x \in \text{out}(S)} \{ \bar{x} \varphi \mid \varphi \in \text{traces}(t_x) \} \). The remainder of the proof is identical to the previous one.

Finally, the theorem below asserts that test cases generated with Algorithm 4.4.1 from a subset of internal-choice traces of a given specification, are always executable on any implementation that behaves like an internal-choice IOLTSs.

**Theorem 4.40.** Let IOLTS \( \bar{s} \) be a specification, let IOTS\( ^{\cap} \) \( i \) be an implementation, and let \( t \) be a test case generated with Algorithm 4.4.1 from \( \bar{s} \) after \( \epsilon \) and I\( \text{Ctraces}(\bar{s}) \). Then \( t \) is an internal-choice test case and hence, it is successfully executable against \( i \).

**Proof.** We know from Propositions 4.38 and 4.39 that \( \text{traces}(t) \subseteq (U \cup \{ \theta \}^+ I) \cup \{ \theta \}^\ast \). Therefore, test case \( t \) is an internal-choice test case. Following Proposition 4.37 IOTS\( ^{\cap} \) \( i \) reaches a quiescent state before an input is provided by \( t \); this input can be accepted by the implementation, which is input enabled in quiescent states. Therefore, \( t \) is executable against implementation \( i \).

Combining Theorem 4.33 with the above theorem yields the following corollary. It states that our test case generation algorithm is sound and exhaustive for the implementation relation \( \text{ico}^{\cap,\cap}_{\text{Ctraces}(\bar{s})} \) for a given specification \( \bar{s} \).

**Corollary 4.41.** Let IOLTS \( \{ S, I, U, \rightarrow, \bar{s} \} \) be a specification. Then

1. a test case obtained with Algorithm 4.4.1 from \( \bar{s} \) after \( \epsilon \) and I\( \text{Ctraces}(\bar{s}) \) is sound for \( \bar{s} \) with respect to \( \text{ico}^{\cap,\cap}_{\text{Ctraces}(\bar{s})} \).

2. the set of all possible test cases that are obtained from Algorithm 4.4.1 from \( \bar{s} \) after \( \epsilon \) and I\( \text{Ctraces}(\bar{s}) \) is exhaustive for \( \bar{s} \) with respect to \( \text{ico}^{\cap,\cap}_{\text{Ctraces}(\bar{s})} \).

### 4.5 Closing Remarks

The internal-choice IOLTSs originally appeared in [WW09] to cater for the asynchrony problem in input-output conformance testing. The main restriction of internal-choice IOLTSs imposed on models used in testing is that the input actions are accepted only at quiescent states. Although such an assumption cannot be made for all systems, internal-choice IOLTSs can still capture systems in practice as discussed in [WW09]. To investigate the effectiveness of the solution based on internal-choice IOLTSs for the \( \text{ico} \) testing framework in the asynchronous setting, we studied the effects of using internal-choice IOLTSs and traces on the notion of conformance testing in the synchronous setting.

In this chapter, we gave unifying intensional and extensional definitions of conformance testing relations for subclasses of IOLTSs, namely input-enabled and internal-choice labeled transition systems. The introduced implementation relations are extensively compared with the standard \( \text{ico} \) relation by restricting either the behavioral models or the set of observations to internal-choice models and traces. It is proven through
various theorems and examples that using internal-choice models as the behavioral models of specifications or implementations has an impact on the testing power in the synchronous execution. However, we established a condition under which the discriminating power of the \textit{ioco} relation is the same as the testing power of the implementation relation that uses internal-choice models: When the IUT behaves as an input-enabled internal-choice IOLTS, the testing power of IOLTS's and that of IOLTSs coincide. With respect to that result, it is sufficient to check only parts of the behavior of a system that are captured by internal-choice traces. In this regard, a constructive solution for developing the test-equivalent IOLTS of a given IOLTS specification was presented. The restriction imposed on the IUT in this case is semantical. To make that result accessible in practice, it is essential to investigate what syntactic conditions imply those semantic conditions. However, regarding the definitions of IOTS and IOTS's, an implementation belonging to IOTS ∩ IOTS is allowed to remain quiescent after any interaction. Such implementations, thus, are likely to be of theoretical interest only. In this sense, we can claim that the testing power of IOLTS's in practice is different from that of IOLTSs.

Subsequently, the effect of using a different set of observations on testing is examined as well. Not surprisingly, we showed that the set of observations used in testing has a direct impact on the result of testing; Checking the internal-choice traces of a specification instead of its suspension traces leads mostly to an implementation relation with a weaker power of discrimination. However, we showed that when the IUT behaves as an input-enabled internal-choice IOLTSs, checking the internal-choice traces of an internal-choice IOLTS specification yields an implementation relation with the same testing power as the \textit{ioco} relation. With respect to that result, to assess the \textit{ioco}-conformity of an implementation behaving as an input-enabled internal-choice IOLTS to an internal-choice IOLTS, it is sufficient to examine the internal-choice traces, a subset of the suspension traces. This way, the number of test cases needed to be examined will be reduced. However, as mentioned above, the conditions imposed on the IUT are semantical; the syntactical condition implying the required semantical conditions still need to be investigated.

The comparison between internal-choice IOLTSs and standard IOLTSs in this chapter was conducted in the synchronous setting. However, the question whether the results of this chapter extend to the asynchronous settings has still to be investigated. We do so in Chapter 6.

We finally modified the test case generation algorithm introduced by Tretmans for the standard \textit{ioco} relation to produce an exhaustive set of sound test cases for the implementation relations previously formalized in this chapter. The set of generated test cases are input enabled which means that they never block any output produced by an implementation. Considering the set of internal-choice traces of a given specification, we prove that the test cases derived from our test case generation algorithm are internal-choice test cases. Thus, they can be successfully executed on implementations that behave as IOTS's. The test case generation algorithm introduced in this chapter can facilitate the development of a unified testing tool for the extended \textit{ioco} family relations.

As a legacy from Tretmans' test case generation algorithm, the set of test cases generated by our algorithm is not minimal. The set of test cases can be reduced while its testing power remains unchanged. In this regard, we can improve our test case generation algorithm. For example our test case generation algorithm allows test cases which perform $\theta$-labeled transitions successively, whereas a test case behaving as $\theta; \theta; t$ has the
same discriminating power as the test case behaving as $\theta; t$. In this way, by restricting the number of successive observations of quiescence to one in a test case, we can remove a large number of redundant test cases from the set of generated test cases from Algorithm 4.4.1 without any impact on its exhaustiveness. This issue is also considered in test case generation algorithms presented in [VT13] and [LG02; Gau07]. Moreover, our test case generation algorithm generates a separate test case for every trace of a given subset of suspension traces, while some of them may be a prefix of other ones. So, as explained in [LG02; Gau07], when a longer trace is considered in constructing a test case, all of its prefix traces are also examined by that test case as well. Following the solution suggested in [LG02; Gau07], we can restructure our test cases such that all test cases relating to prefixes of a certain trace are grouped into one single test case.
Chapter 5

On the Complexity of Input-Output Conformance Testing

One of the obstacles of widespread industrial use of the ioco testing theory is the large number of test cases that have to be executed. In practice, the set of test cases derived from the ioco test case generation algorithm [Tre08] is typically infinite. However, the ioco testing theory is not the only one that suffers from that problem. The selection of a finite subset of an exhaustive test set is one of the problems that has been addressed in the literature [WO80; RC85; Gau95] for several years. In this regard, different types of assumptions have been made about the IUT, which are referred to as *test selection hypotheses* in the literature [Hie+09; Gau95]. The relation between the set of assumptions and the number of test cases is illustrated in Figure 5.1.

Figure 5.1: Framework for the relation between proof and testing [Hie+09]
To establish a test selection hypothesis in the ioco testing framework, in this chapter, we study the complexity of checking ioco, a topic which, as far as we could trace, has not been addressed in the literature. Our study sheds some light on the theoretical boundaries for this popular notion and possible enhancements in its efficiency and efficacy by considering restricted forms of specifications and implementations.

We first show that the upper bound on the complexity of checking ioco is exponential in the size of the model. This is expected due to the trace-based nature of (the intensional definition of) ioco. We show that this exponential complexity bound is indeed tight, by proving that the problem is PSPACE-complete. This means that, unless the complexity class hierarchy from P to PSPACE collapses, the exponential time complexity in deciding ioco is unavoidable in the worst case. Next, we identify a more restricted setting for checking ioco which admits a polynomial time algorithm. In this restricted setting, implementations are still permitted to behave non-deterministically, but specifications must be deterministic. In order to obtain this result, we first give a coinductive (simulation-like) definition of ioco for deterministic specifications and we subsequently show that it can be decided in polynomial time.

Our study is based on the intensional representation of ioco, which allows for defining the exact complexity bounds on checking conformance. The complexity bounds for the intensional representation also hold for the extensional representation, but it remains to be checked under which conditions these bounds can be realized in the practical setting by using test-cases.

Our polynomial time algorithm for checking ioco (for deterministic models) is inspired by [Shu+96], which is based on the reduction of checking ioco into the NHORNSAT problem [DG84]. The coinductive definition of ioco, which is an important means for this result, is akin to the alternating refinement [Alu+98] of Interface Automata [AH01] (see also [AV10; GLM13]). In [VB10], it is shown that for deterministic specifications and implementations, alternating refinement coincides with ioco. In this chapter, we show that our coinductive definition of ioco coincides with ioco for deterministic specifications (and possibly nondeterministic implementations).

In [Hie12], the author proves that testing conformance under asynchronous communication is in general EXPTIME-hard. Then, by restricting to a particular class of models, called observable IOTs, the author gives a polynomial time algorithm for checking conformance under asynchronous FIFO communication. (The problem remains equally hard for observable models under arbitrary asynchronous communication.) Apart from being in the asynchronous setting, the notion of conformance used in [Hie12] differs from ioco (e.g., its theory does not treat quiescence).

Structure of the Chapter. In Section 5.1, we study the complexity of checking ioco. In order to obtain more efficient bounds for checking conformance of deterministic models, we first give a coinductive definition of ioco in Section 5.2 and use this definition in Section 5.3 to show that in this restricted setting, conformance checking is indeed possible in polynomial time. The obtained results in the previous sections are extended in Section 5.4 when the set of internal-choice traces is considered in testing. We conclude the chapter in Section 5.5.
5.1 \textit{ioco} Checking for Nondeterministic Models

In this section, we study the complexity of input-output conformance checking in full generality. We prove that, in the general case, checking \textit{ioco} is PSPACE-complete. To this end, we first show that checking \textit{ioco} is in PSPACE. Subsequently, we show that checking \textit{ioco} is at least as hard as other canonical PSPACE-complete problems and hence, is also PSPACE-complete. We prove these results by means of two polynomial time reductions, respectively, to and from the language inclusion problem for regular expressions (or NFAs); the latter problem is well-known to be PSPACE-complete; see [SM73] for the classical result and [Abd+10; Plo09] for some recent developments.

\textbf{Theorem 5.1.} The problem of checking \textit{ioco} is in PSPACE.

\textit{Proof.} We prove the thesis by showing that the problem of checking \textit{ioco} is polynomial-time reducible to checking a relation between the languages of two NFAs, which is subsequently shown to be in PSPACE.

Assume that, for \(i \in \{1, 2\}\), we have IOLTSs \(A_i\) of the form \((S_i, I, U, \rightarrow_i, \vec{s}_i)\). Our reduction proceeds as follows. We define NFAs \(A'_i = (Q_i, \Sigma_i, \Delta_i, q_i, F)\) as follows:

- \(Q_i = S_i\) is the set of states,
- \(\Sigma = L_s\) is the common alphabet, where \(L = I \cup U\),
- \(\Delta_i = \{ (q, a, q') \mid q \xrightarrow{a} q' \land a \in L \} \cup \{ (q, \epsilon, q') \mid q, q' \in S_i \land q \xrightarrow{\epsilon} q' \} \cup \{ (q, \delta, q) \mid q \in S_i \land \delta(q) \}\) is the transition relation, which is that of the corresponding IOTS to which we add a \(\delta\)-labeled self-loop for every quiescent state,
- \(q_i = \vec{s}_i\) is the initial state, and
- \(F = Q_i\) is the set of final states.

Note that the above given reduction is carried out in time linear in the size of the transition relations of \(A_1\) and \(A_2\). Moreover, observe that \(L(A'_1) = \text{Straces}(A_1)\) and \(L(A'_2) = \text{Straces}(A_2)\).

We now define the conditional language inclusion relation between two NFAs \(A'_1\) and \(A'_2\) with respect to a non-empty set \(R \subseteq \Sigma\), denoted by \(\subseteq_R\), as follows,

\[ L(A'_1) \subseteq_R L(A'_2) \text{ iff } \forall \sigma \in L(A'_1) \cap L(A'_2), \forall x \in R \cdot \sigma x \in L(A'_1) \text{ implies } \sigma x \in L(A'_2) \]

We proceed by showing that \(A_1 \text{ ioco } A_2\) if and only if \(L(A'_1) \subseteq_R L(A'_2)\) with \(R = U \cup \{ \delta \}\). Note that according to the definition of the \textit{ioco} relation, IOLTS \(A_1\) is input enabled. We prove the contraposition of both implications separately.

- Assume that \(A_1 \text{ ioco } A_2\). By definition of the \textit{ioco} relation there is a suspension trace \(\sigma\) in specification \(A_2\) such that for some output \(x\), \(\sigma x \not\in \text{Straces}(A_1)\), but \(\sigma x \not\in \text{Straces}(A_2)\). Since \(L(A'_1) = \text{Straces}(A_1)\) and \(L(A'_2) = \text{Straces}(A_2)\), it follows that \(L(A'_1) \not\subseteq_R L(A'_2)\) for \(R = U \cup \{ \delta \}\).
• Assume \( L(A'_1) \not\subseteq_R L(A'_2) \) for \( R = U \cup \{ \delta \} \). Then there is a word \( \sigma \in L(A'_1) \cap L(A'_2) \) and an output \( x \) such that \( \sigma x \in L(A'_1) \) but \( \sigma x \not\in L(A'_2) \). Since \( L(A'_1) = \text{Straces}(A_1) \), we have that \( \sigma x \in \text{Straces}(A_1) \). Moreover, regarding \( L(A'_1) = \text{Straces}(A_1) \), we find from \( \dagger \) that \( \sigma x \not\in \text{Straces}(A_2) \) while \( \sigma \in \text{Straces}(A_2) \). The last two observations result in \( A_1 \noot A_2 \) which was to be shown.

Now, to show that the problem of checking the \( \noot \) relation between arbitrary IOTS \( A_1 \) and IOLTS \( A_2 \) is in PSPACE, it is sufficient to show that the problem of checking the conditional language inclusion between two NFAs \( A'_1 \) and \( A'_2 \) with respect to a non-empty set \( R \subseteq \Sigma \) is in PSPACE, which is shown below.

The proof of the above thesis proceeds in the same lines of reasoning of the proof of the complexity of the language inclusion problem between NFAs. Note that by choosing \( R = \Sigma \), the language inclusion problem between NFA \( A'_1 \) and NFA \( A'_2 \) is obtained from the conditional language inclusion problem with respect to \( R \). We define NFA \( A''_2 \) as the complementary automaton of NFA \( A'_2 \) with respect to \( R \) such that \( L(A''_2) = \{ \sigma x | \sigma \in L(A'_2) \land x \in R \land \sigma x \not\in L(A'_2) \} \). Note that \( (A''_2) = \) is exponentially bigger than NFA \( A'_2 \). The complementary automaton \( L(A''_2) \) can be constructed by a subtle modification in subset construction algorithm as the standard algorithm for determinization of NFA \( A'_2 \): for every state \( p \subseteq Q_2 \) in the determinizing algorithm of NFA \( A'_2 \), for any \( x \in R \), pair \( (p, x, P_2) \), with \( P_2 \not\subseteq Q_2 \), is added to the transition relation of NFA \( A''_2 \), when there are no \( p \in P \) and \( p' \in Q_2 \) such that \( (p, x, p') \in \Delta_2 \). State \( P_2 \) is the only final state in \( A''_2 \).

Clearly, \( A'_1 \subseteq_R A'_2 \) if and only if \( L(A'_1) \cap L(A''_2) = \emptyset \). Therefore, the problem of checking if \( A'_1 \subseteq_R A'_2 \) is reduced to check the emptiness problem of \( L(A'_1 \times A''_2) \). The emptiness problem can be checked in linear time. NFA \( A''_2 \) and the product automaton \( L(A'_1 \times A''_2) \) cannot be simply constructed and then test it for emptiness. Instead, analogous to the language inclusion problem, the product automaton \( A'_1 \times A''_2 \) is built on-the-fly [Abd+10] and the emptiness of its language is checked. A state in the product automaton \( A'_1 \times A''_2 \) is a pair \((q, P)\) with \( q \in Q_1 \) and \( P \) is a state in NFA \( A''_2 \). A state \((q, P)\) in the product automaton is accepting if and only if \( q \in F_1 \) and \( P \) is also the accepting state in \( A''_2 \), i.e., \( P = P_F \). Whenever the non-emptiness algorithm wants to move from state \( t \) of the product automaton \( A'_1 \times A''_2 \) by a symbol in \( \Sigma \), the destination state \( t' \) is constructed and state \( t \) is not kept in memory anymore [MS72, Theorem 2.2]. Therefore, at each step the algorithm needs to keep just two states of \( A'_1 \times A''_2 \) in memory. This yields a polynomial space algorithm.

We next establish that the problem of checking \( \noot \) is in fact PSPACE-complete.

**Theorem 5.2.** The problem of checking \( \noot \) is PSPACE-complete.

**Proof.** We prove the thesis by providing a linear reduction of the PSPACE-complete language inclusion problem for regular expressions to checking \( \noot \). Every regular expression can be translated linearly to a language equivalent NFA, following Kleene’s theorem.
5.1. \textit{ioco} Checking for Nondeterministic Models

In particular, we may assume that the language-equivalent NFA of a regular expression has one initial and one final state, all states are reachable from the initial state and can reach the final state, there is no incoming transition to the initial state and no outgoing transition from the final state (e.g., by applying Thompson’s algorithm for converting regular expressions to NFAs, which has a linear time complexity in the size of a given regular expression [Tho68]).

Formally, let $RE_1$ and $RE_2$ be two regular expressions over alphabet $\Sigma$ and assume that $A_1$ and $A_2$ are the language-equivalent NFAs for $RE_1$ and $RE_2$. The inclusion problem of regular expressions of $RE_1$ and $RE_2$ is equivalent to the problem whether $L(A_1) \subseteq L(A_2)$. As stated above, we may assume that NFA $A_j$ with $j \in \{1, 2\}$ is of the form $(Q_j, \Sigma, \Delta_j, q_j, \{f_j\})$. We define IOLTSs $A'_j = (S_j, \{i\}, \Sigma, \rightarrow, \bar{s}_j)$ for $j \in \{1, 2\}$ as follows:

- $S_j = Q_j$,
- $i \notin \Sigma$ and $\Sigma$ represent the only input action and the outputs, respectively,
- $\rightarrow_j = \{(q, a, q') \mid (q, a, q') \in \Delta_j \land a \in \Sigma\} \cup \{(q, \tau, q') \mid (q, \tau, q') \in \Delta_j\} \cup \{(q, i, q) \mid q \in Q_j\}$, i.e., the transition relation is that of the corresponding automaton to which we’ve added i-labeled self-loops for each and every state,
- $\bar{s}_j = q_j$.

Note that the two IOLTSs $A'_1$ and $A'_2$ obtained from the above reduction are input-enabled, because $\{i\} \subseteq \text{Sinit}(s)$ for all $s$ in both $A'_1$ and $A'_2$. Moreover, the accepting states in $A_1$ and $A_2$ are the only quiescent states in $A'_1$ and $A'_2$. We proceed to show that language inclusion of $A_1$ in $A_2$ can be decided by checking the \textit{ioco} conformance of $A'_1$ as the implementation to $A'_2$ as the specification; that is, we prove $L(A_1) \subseteq L(A_2)$ if and only if $A'_1 ioco A'_2$. The contraposition of each implication is again proved separately.

- Assume that $L(A_1) \not\subseteq L(A_2)$. Thus, there is a word $\sigma \in \Sigma^*$ such that $\sigma \not\in L(A_1)$ but $\sigma \in L(A_2)$. Therefore, the accepting state $f_1$ in NFA $A_1$ is reachable after $\sigma$. By construction, state $f_1$ in $A'_1$ is quiescent. Thus, $\delta \in \text{out}(A'_1$ after $\sigma$). We distinguish two cases.
  - Suppose there is a state in automaton $A_2$ which is reachable after $\sigma$. Thus, $\sigma \in \text{Traces}(A'_2)$. Since $\sigma \not\in L(A_2)$, we know that $A_2$ does not reach its accepting state $f_2$ after $\sigma$. Therefore, $\delta \not\in \text{out}(A'_2$ after $\sigma$). Since $\delta \in \text{out}(A'_1$ after $\sigma$) and $\sigma \in \text{Traces}(A'_2)$, we find that $A'_1 ioco A'_2$, which was to be shown.
  - Assume there is no state in automaton $A_2$ that is reachable after $\sigma$. Then also $\sigma \not\in \text{Traces}(A'_2)$. Let $\rho x \in \Sigma^*$ be a prefix of $\sigma$ such that $\rho \in \text{Traces}(A'_2)$ but $\rho x \not\in \text{Traces}(A'_2)$. Note that such a prefix must exist. Since $\sigma \in \text{Traces}(A'_1)$, we find that $\rho x \in \text{Traces}(A'_1)$. Therefore $A'_1 ioco A'_2$, which was to be shown.

- Assume that $A'_1 ioco A'_2$. Thus, there is a $\sigma \in \text{Traces}(A'_1) \cap \text{Traces}(A'_2)$ and an output $x \in \Sigma \cup \{\delta\}$ such that $\sigma x \in \text{Traces}(A'_1)$ but $\sigma x \not\in \text{Traces}(A'_2)$. We first define the projection operator $\downarrow$ over the sequences in $\Sigma \cup \{i, \delta\}$. Let $\gamma \in (\Sigma \cup \{i, \delta\})^*$ and $a \in \Sigma \cup \{i, \delta\}$. Then $(\gamma a)_i = (\gamma)_i a$ when $a \in \Sigma$, and $(\gamma a)_i = (\gamma)_i$ otherwise.
Since, by construction, only transitions labeled with an action in $\Sigma$ invoke state changes in IOLTS $A'_1$, for all $\gamma$ we have that $(A'_1 \text{ after } \gamma) = (A'_1 \text{ after } \gamma_1)$, and similarly for $A'_2$ we have that $(A'_2 \text{ after } \gamma) = (A'_2 \text{ after } \gamma_1)$. Therefore, without loss of generality we may assume that $\sigma x = \sigma_1 x$, i.e., $\sigma \in \Sigma^*$. We distinguish two cases, based on the type of $x$.

- Assume that $x \in \Sigma$. Since $\sigma x \in \text{Straces}(A'_1) \cap \Sigma^*$, there is a state in $A_1$ that is reachable after $\sigma x$. Since the accepting state $f_1$ in $A_1$ is reachable from every state in $A_1$, there must be a $\rho \in \Sigma^*$ such that $\sigma x \rho \in L(A_1)$. From $\sigma x \notin \text{Straces}(A'_2)$ and $\sigma x \in \Sigma^*$ we can deduce that no state in $A_2$ can be reached via the word $\sigma x$. Consequently, the extended word $\sigma x \rho$ is also not accepted by $A_2$, i.e., $\sigma x \rho \notin L(A_2)$. Since $\sigma x \rho \in L(A_1)$, we conclude that $L(A_1) \nsubseteq L(A_2)$.

- Assume that $x \notin \Sigma$; it then follows that $x = \delta$. Thus, $\delta \in \text{out}(A'_1 \text{ after } \sigma)$. By our construction, a $\delta$-labeled transition is enabled only at state $f_1$ in $A'_1$ and state $f_2$ in $A'_2$. From this, it follows that word $\sigma$ is accepted by $A_1$, i.e., $\sigma \in L(A_1)$. Following a similar line of reasoning, we conclude from $\delta \notin \text{out}(A'_2 \text{ after } \sigma)$ that the word $\sigma$ is not accepted by $A_2$, i.e., $\sigma \notin L(A_2)$. But then $L(A_1) \nsubseteq L(A_2)$, which we needed to show.

Since the reduction we used is linear in the size of $A_1$ and $A_2$ and since checking ioco conformance is in PSPACE (Theorem 5.1), it follows that checking ioco conformance is PSPACE-complete.

Example 5.3. Consider automata $A_1$ and $A_2$ over $\Sigma = \{a\}$ depicted in Figure 5.2. Automata $A_1$ and $A_2$ accept regular languages $a$ and $aa^*$, respectively. Thus, $L(A_1) \subset L(A_2)$. Now consider the IOLTS’s $A'_1$ and $A'_2$ in Figure 5.2 with $\{a\}$ as the set of outputs and $\{i\}$ as the set of inputs. Observe that IOLTS’s $A'_1$ and $A'_2$ can be obtained from $A_1$ and $A_2$ according to the reduction algorithm presented in the proof of Theorem 5.2. Both models are input enabled because they have a transition labeled with $i$ as the only input action at every state. The set $\text{Straces}(A'_2)$ is given by regular expression $(i^*)| (i^*a(i|i)^*(\delta|ii)^*)$. Since $\text{out}(A'_1 \text{ after } \sigma) \subseteq \text{out}(A'_2 \text{ after } \sigma)$ for all $\sigma \in \text{Straces}(A'_2)$, it is clear that $A'_1 \text{ioco A}'_2$.
5.2 A Coinductive Definition of ioco

In the previous section, we showed that checking \(\text{ioco}\) is in general inefficient and requires exponential time (in the size of the specification) by proving that the problem is PSPACE-complete. In the next section, we show that checking \(\text{ioco}\) can be performed in polynomial time when specifications are deterministic. To prove this result, we first show that checking \(\text{ioco}\) reduces to checking a simulation-like preorder which we call coinductive \(\text{ioco}\). This preorder closely resembles alternating refinement [Alu+98] for Interface Automata [AH01].

Definition 5.4 (Coinductive ioco). Let IOLTS \(\langle S, I, U, \rightarrow, s, \delta \rangle\) be a specification, and let IOTS \(\langle Q, I, U, \rightarrow, r \rangle\) be an implementation. A binary relation \(R \subseteq Q \times \mathcal{P}(S)\) is called a coinductive \(\text{ioco}\) relation from \(r\) to \(s\) when \((r, \{s\}) \in R\) and for each \((q, p) \in R\), then

- (Input simulation) if \(a \in \text{Sinit}(P) \cap I\), then \((q \text{ after } a) \neq \emptyset\) and for all \(q' \in q \text{ after } a\), we have \((q', P') \in R\) with \(P' = P\) after \(a\).
- (Output simulation) if \(q \xrightarrow{a} q'\), and \(a \in U \cup \{\delta\}\), then \(a \in \text{out}(P)\) and we have \((q', P') \in R\) with \(P' = P\) after \(a\).

We write \(r \leq s\), when there exists a coinductive \(\text{ioco}\) relation relating \(r\) to \(s\).

If the intent is clear from the context, we will simply say that a relation is a coinductive \(\text{ioco}\) relation rather than a coinductive \(\text{ioco}\) relation from \(r\) to \(s\). Note that when the specification \(s\) is deterministic, the reachable set of states after executing any trace is at most singleton. We thus for the sake of simplicity, we use \((q, p)\) to show \((q, \{p\}) \in Q \times \mathcal{P}(S)\) when specification \(s\) is deterministic.

The following theorem is the main result of this section.

Theorem 5.5. Coinductive \(\text{ioco}\) and \(\text{ioco}\) coincide.

Before giving the proof of the theorem, we need to show the correctness of the lemma given below.

Lemma 5.6. Let IOLTS \(\langle S, I, U, \rightarrow, s, \delta \rangle\) be a specification, and let IOTS \(\langle Q, I, U, \rightarrow, r \rangle\) be an implementation over language \(L = I \cup U\). Let \(R \subseteq Q \times \mathcal{P}(S)\) be a coinductive \(\text{ioco}\) relation, and let \(\sigma \in \text{Straces}(s) \cap \text{Straces}(r)\) with length \(n \geq 1\). Then, \((q, p) \in R\) for \(p = (\overline{s} \text{ after } \sigma)\) and all \(q \in (\overline{r} \text{ after } \sigma)\).

Proof. Because \(R\) is a coinductive \(\text{ioco}\) relation, we have \((\overline{r}, \{s\}) \in R\). We proceed with an induction on the length of \(\sigma\).

- For the base case, assume that \(\sigma \in L_\overline{s}\) is a suspension trace of length 1. We distinguish two cases. Suppose that \(\sigma \in I\). Following the input simulation condition with \((\overline{r}, \{s\}) \in R\), we immediately find that \((q, p) \in R\) for \(p = (\overline{s} \text{ after } \sigma)\) and \(q \in (\overline{r} \text{ after } \sigma)\). Suppose \(\sigma \in U \cup \{\delta\}\). It follows from the output simulation condition together with \((\overline{r}, \{s\}) \in R\) that \((q, p) \in R\) for \(p = (\overline{s} \text{ after } \sigma)\) and all \(q \in (\overline{r} \text{ after } \sigma)\). Both cases lead to the desired result.
Proof of Theorem 5.5.

The proof of each implication is given separately.

Chapter 5. On the Complexity of Input-Output Conformance Testing

Now, we are in the position to give the proof of Theorem 5.5.

\[ p = (s \text{ after } \sigma) \text{ and arbitrary } q \in (\bar{r} \text{ after } \sigma). \]

\[ \square \]

\textbf{Proof of Theorem 5.5.} The proof of each implication is given separately.

- We suppose that \( \bar{r} \text{ ioco } \bar{s} \), then we show that there is no binary relation \( R \) such that \((\bar{r}, \{\bar{s}\}) \in R \) and \( R \) is such that the input and output simulation conditions hold for all pairs \((q, p) \in R \). By definition of \text{ ioco }, we know that there exists a sequence \( \sigma \in \text{Straces}(\bar{r}) \cap \text{Straces}(\bar{s}) \) and there exists an output \( x \) such that \( \sigma x \in \text{Straces}(\bar{r}) \) but \( \sigma x \notin \text{Straces}(\bar{s}) \). We distinguish two cases; \( \sigma = \epsilon \) and \( \sigma \in L_\delta^+ \).

  - Suppose that \( \sigma = \epsilon \). Clearly, the pair \((\bar{r}, \{\bar{s}\})\) violates the output simulation condition, because \( \bar{r} \Rightarrow q \), while \( \bar{s} \not\Rightarrow x \). Therefore, there can be no relation \( R \) that simultaneously satisfies the required simulation properties and \((\bar{r}, \{\bar{s}\}) \in R \).

  - Suppose \( \sigma \in L_\delta^+ \). Towards a contradiction, assume that there is a coinductive \text{ ioco } relation \( R \). Thus, \((\bar{r}, \{\bar{s}\}) \in R \). It follows from \( \sigma x \in \text{Straces}(\bar{r}) \) that there is some \( q \in (\bar{r} \text{ after } \sigma) \) such that \( x \in \text{out}(q) \). From \( \sigma \in \text{Straces}(\bar{s}) \) we find that there is a \( p \in \text{P}(S) \) such that \( p = (\bar{s} \text{ after } \sigma) \neq \emptyset \). Because \( \sigma x \notin \text{Straces}(\bar{s}) \), we find that \( x \notin \text{out}(p) \). From lemma 5.6, we obtain that \((q, p) \in R \). However, the pair \((q, p) \) violates the output simulation condition since \( q \Rightarrow x \) but \( x \notin \text{out}(p) \). This contradicts the assumption that there is a coinductive \text{ ioco } relation \( R \).

- We suppose that \( \bar{r} \text{ ioco } \bar{s} \). We construct the relation \( R = \{(\bar{r}, \{\bar{s}\})\} \cup \{(q, p) \mid \exists \sigma \in L_\delta^+ \bullet \bar{r} \Rightarrow q \land p = \bar{s} \text{ after } \sigma \} \). We proceed to show that \( R \) is a coinductive \text{ ioco } relation. Clearly, \((\bar{r}, \{\bar{s}\}) \in R \). Hence, it suffices to show that for an arbitrary pair \((q, p) \in R \), the input and output simulation conditions are met. We assume arbitrary pair \((q, p) \in R \). Thus, there exists \( \sigma \in L_\delta^+ \) such that \( \bar{r} \Rightarrow q \) and \( p = \bar{s} \text{ after } \sigma \). Because \( \bar{r} \) is input enabled, \( q \) has a matching (weak) transition for any input action performed by \( p \), i.e., for all \( a \in I \cap \text{Sinit}(p) \), \( q \text{ after } a \neq \emptyset \). By the definition of \( R \), we find that \((q', p') \in R \) for an action \( a \in I \) such that \( p' = p \text{ after } a \) and any \( q \Rightarrow q' \). Thus, \((q, p) \) satisfies the input simulation condition. Using \( \bar{r} \text{ ioco } \bar{s} \), by construction of \( R \), we know that \( \text{out}(q) \subseteq \text{out}(p) \). Combining this observation with the definition of \( R \) results in \((q', p') \in R \) for all actions \( a \in U \cup \{\delta\} \) for which \( p' = p \text{ after } a \) and \( q \Rightarrow q' \). Thus, \((q, p) \) also satisfies the output simulation condition. Hence, the pair \((q, p) \) fulfills both the input and the output simulation conditions. Since \( R \)
satisfies both simulation conditions and \((\bar{r}, \bar{s}) \in R\), we find that \(R\) is a coinductive \(\text{ioco}\) relation.

Following Theorem 5.5, we say that a coinductive \(\text{ioco}\) relation \(R\) is a witness for \(\text{ioco}\) \(\bar{s}\).

**Example 5.7.** Consider the IOLTS's \(\bar{s}\) and \(\bar{r}\) presented in Figure 5.3. The IOLTS \(\bar{s}\) is a specification of a vending machine which sells tea. After receiving a coin, it either delivers tea or refunds the coin. The IOTS \(\bar{r}\) is a formal model of a possible implementation of this vending machine. Upon receiving a coin, machine \(\bar{r}\) chooses nondeterministically between serving tea or refunding the coin. We define the binary relation \(R = \{(\bar{r}, \bar{s}), (r_1, s_1), (r_2, s_1), (r_3, s_2), (r_4, s_3)\}\). Clearly, for all pair of states \((q, p) \in R\), the two input and output simulation conditions presented in Definition 5.4 are satisfied. Thus, the relation \(R\) is a coinductive \(\text{ioco}\) relation and it is also a witness for \(\text{ioco}\) \(\bar{s}\).

Now, consider the IOTS \(\bar{i}\) depicted in Figure 5.3. The IOTS \(\bar{i}\) is a formal model of an implementation of a malfunction vending machine. After receiving a coin, it either delivers tea, refunds the coin or does nothing. It is clearly obtained that \(\bar{i} \text{ioco} \bar{s}\), because \(\text{out}(\bar{i} \text{ after coin}) \not\subseteq \text{out}(\bar{s} \text{ after coin})\). Therefore, there is no binary relation from \(\bar{i}\) to \(\bar{s}\) such that \((\bar{i}, \bar{s}) \in R\) and the two input and output simulation conditions hold for any pair \((q, p) \in R\). However, for the sake of contradiction, we assume that there is a relation \(R'\) such that \((\bar{i}, \bar{s}) \in R'\) and \(R'\) is such that for any \((q, p) \in R'\), the two conditions in Definition 5.4 hold. Due to the input simulation condition, \((\bar{i}, \bar{s}) \in R'\) implies that \((i_1, s_1) \in R'\) as well. We know from the properties of \(R'\), that \(s_1\) has to simulate all the outputs produced by \(i_1\), while observing quiescence is not possible at \(s_1\) (not even via an internal transition \(i_1\) that can reach to a quiescent state). Therefore, \((i_1, s_1)\) violates the output simulation condition which contradicts the assumption that all pairs in \(R'\) respect the output simulation condition.

### 5.3 \(\text{ioco}\) Checking of Deterministic Specifications

In this section, we give a polynomial-time algorithm for deciding the coinductive \(\text{ioco}\) relation defined in the previous section. The results obtained in the remainder of this section can be adapted in a straightforward manner to some other conformance relations.
in the io-co family, such as io-co_u [BRT03]. Our algorithm is inspired by [Shu+96] and is based on the reduction of checking io-co into the NHORSAT problem [DG84].

5.3.1 NHORSAT Problem

The satisfiability problem for Boolean formulas is a typical (in fact, the first identified) NP-complete problem. In a restricted setting, however, the problem becomes decidable in polynomial time.

Definition 5.8 ((N)HORSAT). A boolean clause (a disjunction of literals) containing at most one positive literal is called a Horn clause. We call the conjunction of Horn clauses a Horn formula. The satisfiability of a Horn formula is known as HORSAT. Similarly, checking the satisfiability of a conjunction of clauses containing of at most one negative literal is called NHORSAT.

The size of a (N)HORSAT instance is defined as the total number of occurrences of literals in the given formula. It is well-known that (N)HORSAT is decidable in polynomial time in the size of the (N)HORSAT instance [DG84].

5.3.2 Reducing IOCO to NHORSAT

Throughout this section, we assume that we have an IOTS \( \langle Q, I, U, \rightarrow, \overline{r} \rangle \) and a deterministic IOLTS \( \langle S, I, U, \rightarrow_s, \overline{s} \rangle \) over language \( L = I \cup U \). We assume \( p, p', p'' \) are states in \( S \) and \( q, q', q'' \) are states in \( Q \). The algorithm, which we will present shortly, intuitively uses the following encoding:

1. a positive literal \( X_{qp} \) models that \( q \) is (purportedly) related to \( p \) by a coinductive io-co relation,

2. a negative literal \( \overline{X_{qp}} \) models that the pair \( (p, q) \) cannot be in a coinductive io-co relation, and

3. an implication clause \( X_{qp} \Rightarrow X_{q'p'} \), which is a shorthand for \( \overline{X_{qp}} \lor X_{q'p'} \), models that the pair \( (p, q) \) can be in a coinductive io-co relation only if \( (q', p') \) is in the same relation.

The reduction of checking for a coinductive io-co relation to NHORSAT is presented in Algorithm 5.3.1: this algorithm constructs a negative Horn formula \( F \) such that \( F \) is
5.3. ioco Checking of Deterministic Specifications

satisfiable if and only if there exists a coinductive ioco relation $R$ from $\bar{r}$ to $\bar{s}$.

**Algorithm 5.3.1: ioco-NHORN($\bar{s}, \bar{r}$)**

\[
F \leftarrow X_{\bar{r}} \bar{s} \quad \text{// Positive literal $X_{\bar{r}} \bar{s}$ is added to Formula $F$.
C \leftarrow \{X_{\bar{r}}\} \quad \text{// Set of unprocessed variables
V \leftarrow \emptyset \quad \text{// Set of processed variables

while $C \neq \emptyset$ do

Choose $X_{qp} \in C$

$V \leftarrow V \cup \{X_{qp}\}$

$C' \leftarrow \emptyset$

for each $a \in \text{init}(p) \cap I$ do

if $(q \text{ after } a) \neq \emptyset$ then

Choose $p' \in (p \text{ after } a)$ \quad // Due to determinism, $|p \text{ after } a| = 1$

$F \leftarrow F \land \bigwedge_{q' \in (q \text{ after } a)} (X_{qp} \Rightarrow X_{q'p'})$ \quad // Input simulation condition

$C' \leftarrow C' \cup \{X_{q'p'} \mid q' \in q \text{ after } a\}$ \quad // Add unprocessed variables

else

$F \leftarrow F \land \neg X_{qp}$ \quad // Violation of input simulation

end if

end for

for each $a \in \text{out}(q)$ do

if $a \in \text{out}(p)$ then

Choose $p' \in (p \text{ after } a)$ \quad // Due to determinism, $|p \text{ after } a| = 1$

$F \leftarrow F \land \bigwedge_{q' \in (q \text{ after } a)} (X_{qp} \Rightarrow X_{q'p'})$ \quad // Output simulation condition

$C' \leftarrow C' \cup \{X_{q'p'} \mid q' \in q \text{ after } a\}$ \quad // Add only new variables

else

$F \leftarrow F \land \neg X_{qp}$ \quad // Violation of Output simulation

end if

end for

$C \leftarrow (C \cup C') \setminus V$

return $(F)$ \quad // The final negative HORN formula

The algorithm takes an implementation $\bar{r}$ and a deterministic specification $\bar{s}$ as input. We assume that for $\bar{r}$, the generalized transition relation $\Rightarrow$ from $\rightarrow$ of $\bar{r}$ has been computed. This requires a pre-processing step of $\bar{r}$, involving a transitive closure computation, see e.g., [KS90]. Computing $\Rightarrow$ can be done in polynomial time.

It is easy to see that the algorithm terminates. In each iteration, of the outer loop, the set $V \subseteq \{X_{qp} \mid q \in Q, p \in S\}$ strictly increases and $C \subseteq \{X_{qp} \mid q \in Q, p \in S\} \setminus V$ is a loop invariant. The algorithm thus terminates after at most $|Q| \times |S|$ iterations of the outer loop. Since the set of actions $L_\delta$ is finite, termination of the two inner loops is also guaranteed. It is equally easy to see that the formula that is constructed is a NHORNSAT.
Example 5.9. Reconsider IOLTSs  and in Figure 5.3 on page 91. As we concluded in Example 5.7 on page 91,  because, e.g., out(  after coin)  out(. Therefore, the NHORNSAT instance obtained from Algorithm 5.3.1 must be unsatisfiable. The formula  generated by Algorithm 5.3.1 is the following:

\[
X_{i\bar{s}} \land (X_{i\bar{s}} \Rightarrow X_{i\bar{s}}) \land (X_{i\bar{s}} \Rightarrow X_{i\bar{s}}) \land (X_{i\bar{s}} \Rightarrow X_{i\bar{s}}) \land (X_{i\bar{s}} \Rightarrow X_{i\bar{s}})
\]

Indeed, it is easily seen that the obtained formula  is unsatisfiable: for  to be satisfiable,  must be true, which means that  is true, but that means that  is false.

Next, reconsider IOLTS  of Figure 5.3. We know from Example 5.7 that  is coinductive. The formula  generated by Algorithm 5.3.1 is the following:

\[
X_{q\bar{s}} \land (X_{q\bar{s}} \Rightarrow X_{q\bar{s}}) \land (X_{q\bar{s}} \Rightarrow X_{q\bar{s}}) \land (X_{q\bar{s}} \Rightarrow X_{q\bar{s}}) \land (X_{q\bar{s}} \Rightarrow X_{q\bar{s}})
\]

Clearly, the constructed formula is satisfiable: assigning true to all literals is a satisfying assignment.

5.3.3 Correctness of the Reduction Algorithm

The constructed formula  by Algorithm 5.3.1 has two key properties that together ensure the correctness of our algorithm. First, the existence of a coinductive relation  implies satisfiability of . This follows from the observation that from any coinductive relation  the truth assignment  for  defined by assigning true to every variable  appearing in  for which  and assigning false to all remaining variables in  is a witness to the satisfiability of . Second, satisfiability of  implies the existence of a coinductive relation . In a nutshell, this follows from the observation that for any given satisfying assignment  of , the binary relation  defined by  and  is a coinductive relation. We first prove these two properties, and then state our main theorem claiming the correctness of the algorithm.

Proposition 5.10. Let  be a deterministic IOLTS and let  be an arbitrary IOLTS. Let  be the NHORNSAT instance from Algorithm 5.3.1. If  then  is satisfiable.

The correctness of the above-given proposition results from the following lemma. This lemma essentially states that the presence of a negative literal  in formula  indicates that the pair  can never be related by a coinductive relation.

Lemma 5.11. Let  be the formula obtained from Algorithm 5.3.1, and let  be an arbitrary variable. If  contains the literal , then no coinductive relation  for which  exists.

Proof. Towards a contradiction, assume there is a coinductive relation  such that  exists. In our algorithm, the literal  is only added to  under one of the following two conditions:
1. there is an input action \( a \in \text{init}(p) \) while \( q \) after \( a = \emptyset \),
2. there is an output action \( a \in \text{out}(q) \) while \( a \notin \text{out}(p) \).

We first assume that \( X_{qp} \) is generated because of the first case, i.e., there is an input \( a \in \text{init}(p) \) for which \( q \) after \( a = \emptyset \). Then the pair \( (q, p) \in R \) does not meet the input simulation condition of Definition 5.4, contradicting the fact that the pair \( (q, p) \) can be in a coinductive \textit{ioco} relation \( R \). Next, assume that \( X_{qp} \) is generated because of the second case. Following the same line of reasoning, the presence of \( (q, p) \in R \) violates the output simulation condition, contradicting that \( R \) is a coinductive \textit{ioco} relation.

Next, we return to proving Proposition 5.10.

Proof of Proposition 5.10. Consider a coinductive \textit{ioco} relation \( R \subseteq Q \times S \). Let \( \nu \) be a truth assignment for the variables in \( F \) defined as follows:

\[
\nu(X_{qp}) = \begin{cases} 
\text{true} & \text{if } (q, p) \in R \\
\text{false} & \text{otherwise}
\end{cases}
\]

Since \( (\tilde{r}, \tilde{s}) \in R \), we know that the single literal clause \( X_{\tilde{r} \tilde{s}} \) evaluates to \textit{true}. Next, consider the other two types of clauses that are introduced in formula \( F \): single negative literal clauses and implication clauses.

- **Clauses of the form \( X_{qp} \).** Due to Lemma 5.11 we have \( (q, p) \notin R \) whenever the negative literal clause \( \overline{X_{qp}} \) is added to \( F \) in line (iii) or line (vi). By the definition of \( \nu \) we then have \( \nu(X_{qp}) = \text{false} \). Consequently, a negative literal clause \( \overline{X_{qp}} \) in \( F \) evaluates to \textit{true}.

- **Clauses of the form \( X_{qp} \Rightarrow X_{q'p'} \).** We distinguish the cases when \( (q, p) \notin R \) and \( (q, p) \in R \).

  - Assume that \( (q, p) \notin R \). By the definition of \( \nu \), we have \( \nu(X_{qp}) = \text{false} \). Then the clause \( X_{qp} \Rightarrow X_{q'p'} \) immediately evaluates to \textit{true} under \( \nu \).

  - Suppose that \( (q, p) \in R \). Thus \( \nu(X_{qp}) = \text{true} \). Therefore the clause \( X_{qp} \Rightarrow X_{q'p'} \) evaluates to \textit{true} only if \( \nu(X_{q'p'}) = \text{true} \). The implication clause \( X_{qp} \Rightarrow X_{q'p'} \) in Algorithm 5.3.1 is added to \( F \) in line (ii) when there is some input \( a \in \text{init}(p) \) or in line (v) when there is some output \( a \in \text{out}(q) \) for which \( q' \in q \) after \( a \) and \( p' \in p \) after \( a \). From these observations, and the fact that \( R \) is a coinductive \textit{ioco} relation it follows that \( (q', p') \in R \). But then, by the definition of \( \nu \), we have \( \nu(X_{q'p'}) = \text{true} \), which was to be shown.

As a result, implication clauses in \( F \) of the form \( X_{qp} \Rightarrow X_{q'p'} \) evaluate to \textit{true}.

Since there are no other types of clauses in \( F \), formula \( F \) evaluates to \textit{true} under \( \nu \). \( \square \)

The proposition below formalizes the second property of algorithm \textit{ioco-NHORN}.

**Proposition 5.12.** Let \( \langle S, I, U, \rightarrow, s, \tilde{s} \rangle \) be a deterministic IOLTS and let \( \langle Q, I, U, \rightarrow, \tilde{r} \rangle \) be an arbitrary IOTS. Let \( F \) be the \textbf{NHORSAT} instance from Algorithm 5.3.1. If \( F \) is satisfiable, then \( \tilde{r} \preceq \tilde{s} \).
Observe that this means that formula $F$ is satisfiable if and only if there is a coinductive relation, stating that our reduction algorithm for checking $\text{ioco}$ (constructs formula $F$, which is of size $O$.

Let $\text{Theorem 5.13.}$

We next analyze the complexity of Algorithm 5.3.1. Since $\text{NHORNSAT}$ is decidable in linear time [DG84], proving that we can decide that a possibly non-deterministic implementation conforms to a deterministic specification in polynomial time only requires showing that the Negative Horn formula $F$ can be constructed in polynomial time.

**Theorem 5.14.** Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be a deterministic IOLTS over language $L = I \cup U$ and let $\langle Q, I, U, \rightarrow, \bar{r} \rangle$ be an arbitrary IOTS for which $\Rightarrow$ has been computed. Algorithm 5.3.1 constructs formula $F$, which is of size $O(|S| \times |Q|^2 \times |L_\delta|)$ in time $O(|S| \times |Q|^2 \times |L_\delta|)$.

**Proof.** Let $\nu$ be a truth assignment such that formula $F$ evaluates to $\text{true}$. We construct a binary relation $R \subseteq Q \times S$ as follows:

$$R = \{ (q, p) \mid \text{variable } X_{qp} \text{ occurs in } F \text{ and } \nu(X_{qp}) = \text{true} \}$$

We proceed by showing that $R$ is a coinductive ioco relation. Clearly, since the single literal $X_{q\bar{s}}$ occurs in $F$ and $F$ is satisfiable, we have $\nu(X_{q\bar{s}}) = \text{true}$. By definition, we then have $(\bar{r}, \bar{s}) \in R$.

Let $(q, p) \in R$ be an arbitrary pair. By definition, this means that $\nu(X_{pq}) = \text{true}$. Observe that this means that formula $F$ cannot contain the single negative literal $\overline{X_{qp}}$.

We next show that the pair $(q, p) \in R$ meets both the input and the output simulation conditions:

- **Input simulation.** Suppose that $p \xrightarrow{a} p'$ for some $a \in I$. Since $F$ does not contain the negative literal $\overline{X_{qp}}$, we know that $q$ after $a \neq \emptyset$ (line iii). Therefore, $F$ contains implication clauses of the form $X_{qp} \Rightarrow X_{q'p'}$ where $q' \in q$ after $a$ and $p' \in p$ after $a$. Since $F$ evaluates to true under $\nu$, also $X_{qp} \Rightarrow X_{q'p'}$ evaluates to true under $\nu$. Since $q' \in q$ after $a$ is chosen arbitrarily, we find that $\nu(X_{q'p'}) = \text{true}$ for all $q' \in q$ after $a$. Then by construction, $(q', p') \in R$ for all $q' \in q$ after $a$.

- **Output simulation.** Suppose that $q \Rightarrow q'$ for some $a \in U \cup \{\delta\}$. Following the same line of reasoning as in the above case, we find that the pair $(q, p)$ meets the output simulation condition.

\[\square\]

An immediate consequence of the preceding two propositions is the following theorem, stating that our reduction algorithm for checking ioco is sound.

**Theorem 5.13.** Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be a deterministic IOLTS and let $\langle Q, I, U, \rightarrow, \bar{r} \rangle$ be an arbitrary IOTS. Let $F$ be the $\text{NHORNSAT}$ instance from Algorithm 5.3.1. Then $F$ is satisfiable iff $\bar{r}$ ioco $\bar{s}$.

**Proof.** Following Propositions 5.10 and 5.12, we know that the formula $F$ obtained from Algorithm 5.3.1 is satisfiable if and only if there is a coinductive ioco relation. Combined with Theorem 5.5 we find that formula $F$ is satisfiable if and only if $\bar{r}$ ioco $\bar{s}$.

\[\square\]

### 5.3.4 Complexity Analysis

We next analyze the complexity of Algorithm 5.3.1. Since $\text{NHORNSAT}$ is decidable in linear time [DG84], proving that we can decide that a possibly non-deterministic implementation conforms to a deterministic specification in polynomial time only requires showing that the Negative Horn formula $F$ can be constructed in polynomial time.

**Theorem 5.14.** Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be a deterministic IOLTS over language $L = I \cup U$ and let $\langle Q, I, U, \rightarrow, \bar{r} \rangle$ be an arbitrary IOTS for which $\Rightarrow$ has been computed. Algorithm 5.3.1 constructs formula $F$, which is of size $O(|S| \times |Q|^2 \times |L_\delta|)$ in time $O(|S| \times |Q|^2 \times |L_\delta|)$.

96
Proof. To facilitate writing the proof, we first introduce some auxiliary notation. Let $d^i_a$ denote the cardinality of the set of states reachable from a state $t$ after executing an $a$-labeled transition, i.e., $d^i_a = |t$ after $a|$.

The main loop of Algorithm 5.3.1 iterates over the set of variables of the form $X_{qp}$, for $q \in Q$ and $p \in S$. This means there are at most $|Q| \times |S|$ iterations. The complexity of a single iteration is given by the sum of the complexity of the two inner loops (lines i and iv).

Since the size of a clause $\overline{X_{qp}}$ is smaller than the size of an implication clause introduced in line (ii) or line (v), the size of the constructed clause in each iteration of one of the inner loops is bounded from above by $2 \times d^q_a$ for each $a \in I$ for the first inner loop, and $2 \times d^q_a$ for each $a \in U \cup \{\delta\}$. The cumulative size of the generated clauses in the first inner loop is therefore bounded from above by $2 \times \sum_{a \in I} d^q_a$ and the cumulative size of the generated clauses in the second inner loop is bounded from above by $2 \times \sum_{a \in U \cup \{\delta\}} d^q_a$.

Thus, the cumulative size of the clauses added in each iteration of the outer loop is at most $2 \times (\sum_{a \in I} d^q_a + \sum_{a \in U \cup \{\delta\}} d^q_a) = 2 \times \sum_{a \in I} d^q_a$. Assuming that for all $q \in Q$ and all $p \in S$, all variables $X_{qp}$ are inspected, the total size of the $\text{NHORSAT}$ instance is bound from above as follows:

$$\leq \sum_{p \in S} \sum_{q \in Q} (2 \times \sum_{a \in I} d^q_a) \leq 2 \times |S| \times |Q|^2 \times |L_\delta|$$

Observe that at $\dagger$, we used the fact that $d^q_a$ is bounded from above by the size of the state space of $\bar{r}$, i.e., $|Q|$. Hence, the size of formula $F$ is $O(|S| \times |Q|^2 \times |L_\delta|)$. Since we assume that $\Rightarrow$ has been computed, all operations involving $\Rightarrow$, such as after and out(_, ) require constant time. Constructing formula $F$ can therefore also be done in time $O(|S| \times |Q|^2 \times |L_\delta|)$.

The theorem below states the complexity of deciding $\text{ioco}$ for deterministic specifications and possibly non-deterministic implementations.

**Theorem 5.15.** Let $\langle S, I, U, \rightarrow_x, \bar{s} \rangle$ be a deterministic IOLTS and let $\langle Q, I, U, \rightarrow, \bar{r} \rangle$ be an arbitrary IOTS over language $L = I \cup U$. Deciding whether $\text{ioco}\bar{s}$ for deterministic specifications $\bar{s}$ and possibly non-deterministic implementations $\bar{r}$ can be done in $O(|L_\delta| \times |Q|^{2.3727}) + O(|S| \times |Q|^2 \times |L_\delta|)$.

**Proof.** Generating and solving the NHORN formula $F$ obtained from Algorithm 5.3.1 requires $O(|S| \times |Q|^2 \times |L_\delta|)$, this follows from Theorem 5.14 combined with the fact that $F$ can be solved in time linear in the size of $F$. A precondition to the algorithm is that $\Rightarrow$ has been computed from $\rightarrow$. Following [KS90], this can be done in $O(|L_\delta| \times |Q|^{2.3727})$. \hfill $\Box$

When both implementation and specification are deterministic, the time complexity of our algorithm reduces to $O(|S| \times |\rightarrow|)$. Note that in this case, the computation of $\Rightarrow$ only requires augmenting the transition relation with $\delta$ transitions, which can be done in $O(|\rightarrow|)$. 

97
5.4 Conformance Checking for Internal-Choice Traces

In this section, we study the complexity of input-output conformance checking when the set of observation is restricted to the set of internal-choice traces of a given specification. Analogous to the results of Section 5.1, we show that, in the general case, checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ with $a, b \in \{\cap, \cup\}$ (see Definition 4.1 in Chapter 4) is PSPACE-complete for a given specification $s$. Taking the same approach of Section 5.1, we show the validity of the above thesis by two polynomial reductions to and from the language inclusion problem for regular expressions.

Similar to Theorem 5.1, the theorem below states that checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ with $a, b \in \{\cap, \cup\}$ for a given IOLTS $s$, in general, is PSPACE.

**Theorem 5.16.** The problem of checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ with $a, b \in \{\cap, \cup\}$ for a given IOLTS $s$ is in PSPACE.

**Proof.** Analogous to the proof of Theorem 5.1, we show the correctness of the above thesis by means of a polynomial time reduction of the problem of checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ for $a, b \in \{\cap, \cup\}$ to a problem of checking a relation between the languages of two NFAs with $\varepsilon$-moves, which is subsequently to be in PSPACE. To this end, we reuse and customize the reduction which is presented in the proof of Theorem 5.1 such that the languages of the obtained NFAs represent the set of internal-choice traces of a given specification and the IUT.

Assume that, for $i \in \{1, 2\}$, we have IOTSS $A_i$ of the form $(S_i, I, U, \rightarrow_i, s_i)$, over language $L = I \cup U$. Our reduction proceeds is described next. We define NFAs $A_i' = \langle Q_i, \Sigma, \Delta_i, q_i, F \rangle$ as follows:

- $Q_i = S_i$ is the set of states,
- $\Sigma = L \cup \{\varepsilon\}$, where $L$ is partitioned into the two disjoint sets of inputs $I$ and outputs $U$, is the common alphabet,
- $\Delta_i = \{(q, a, q') \mid q \xrightarrow{a} q' \land a \in U\} \cup \{(q, a, q') \mid q \xrightarrow{a} q' \land a \in I \land \delta(q)\} \cup \{(q, \varepsilon, q') \mid q, q' \in S_i \land q \xrightarrow{\tau} q'\} \cup \{(q, \delta, q) \mid q \in S_i \land \delta(q)\}$ is the transition relation.
- $q_i = s_i$ is the initial state, and
- $F = Q_i$ is the set of final states.

Note that the above reduction is carried out in time linear in the size of the transition relations of $A_1$ and $A_2$. Moreover, since only the input transitions of quiescent states in $A_1$ and $A_2$ are added to NFAs $A_1'$ and $A_2'$, respectively, we find that $L(A_1') = \text{ICtraces}(A_1)$ and $L(A_2') = \text{ICtraces}(A_2)$.

The rest of the proof is identical to the proof of Theorem 5.1. ☐

**Example 5.17.** Consider the IOTS's $\bar{r}$ and $\bar{t}$ presented in Figure 5.4 on page 99. The IOTS $\bar{t}$ represents the behavioral specification of a vending machine which sells tea. After receiving a coin, it either delivers tea, refunds the coin, or receives the order of adding sugar.
5.4. Conformance Checking for Internal-Choice Traces

The IOTS $\tilde{r}$ is a formal specification of a possible implementation of this vending machine. Upon receiving a coin, machine $\tilde{r}$ chooses non-deterministically between serving tea or refunding the coin. Since, $\text{out}(\tilde{r} \text{ after } \sigma) \subseteq \text{out}(\tilde{i} \text{ after } \sigma)$ holds for all $\sigma \in \text{ICtraces}(\tilde{i})$, we obtain that $\tilde{r} \text{ioco}^{-1} \tilde{i}$ with $F = \text{ICtraces}(\tilde{i})$. Consider now two NFAs $A_1$ and $A_2$ depicted in Figure 5.5. Those NFAs are constructed from IOLTSs $\tilde{r}$ and $\tilde{i}$, respectively, according to the reduction presented in the proof of Theorem 5.16. The language of NFA $A_1$ is given by

$$L(A_1) \subset L(A_2).$$

The theorem below asserts that the problem of checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ for $a, b \in \{\sqcap, \sqcup\}$ is indeed PSPACE-complete.

**Theorem 5.18.** The problem of checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ for a given IOLTS $s$ with $a, b \in \{\sqcap, \sqcup\}$ is PSPACE-complete.

**Proof.** We prove the above thesis by providing a linear reduction of the language inclusion problem of regular expression to the problem of checking $\text{ioco}_{\text{ICtraces}(s)}^{a,b}$ for $a, b \in \{\sqcap, \sqcup\}$. Similar to the proof of Theorem 5.2, we assume that $A_1$ and $A_2$ are the language-equivalent NFAs for two regular expression $RE_1$ and $RE_2$ over alphabet $\Sigma$, respectively,
Chapter 5. On the Complexity of Input-Output Conformance Testing

according to Kleene’s theorem. Thus, the inclusion problem of regular expressions of \(RE_1\) and \(RE_2\) is equivalent to the problem whether \(L(A_1) \subseteq L(A_2)\). We know form Kleene’s theorem that NFAs \(A_i\) with \(i \in \{1, 2\}\) can be of the form \((Q_i, \Sigma \cup \{\varepsilon\}, \Delta_i, q_i, \{f_i\})\) with a single initial state and one final state without any outgoing transition that is reachable by all other states. With respect to the NFAs \(A_1\) and \(A_2\), we define IOLTSs \(A_i' = \langle S_i, \{i\}, \Sigma, \rightarrow_i, \bar{s}_i\rangle\) for \(i \in \{1, 2\}\). The IOLTS \(A_2' = \langle S_2, \{i\}, \Sigma, \rightarrow_2, \bar{s}_2\rangle\) is defined as follows:

- \(S_2 = Q_2\),
- \(i \notin \Sigma\) and \(\Sigma\) are the only input and the set of output actions respectively,
- \(\rightarrow_2 = \{(q, a, q') \mid (q, a, q') \in \Delta_2 \land a \in \Sigma\} \cup \{(q, \varepsilon, q') \mid (q, \varepsilon, q') \in \Delta_2\} \cup \{(f_2, i, f_2)\}\),
  i.e., the transition relation is that of the corresponding automaton to which we have added \(i\)-labeled self-loops at the final state,
- \(\bar{s}_2 = q_2\).

Note that IOLTS \(A_2'\) is indeed an IOLTS\(^\pi\), since it has an input transition only at its only quiescent state, state \(f_2\). With respect to the implementation relation \(\text{ioco}^{a,b}_{\text{traces}(s)}\) with \(a, b \in \{\Pi, \omega\}\), the IOLTS \(A_1'\) as the behavioral model of the IUT can be either of the form IOTS or IOTSi. In the case \(A_1' \in \text{IOTS}(\{i\}, \Sigma)\), it is constructed in the same way as IOLTS\(^\pi\) \(A_2'\), where its transition relation is that of NFA \(A_1\) to which an \(i\)-labeled self-loop is added only at the final state of NFA \(A_1\) as the only quiescent state in \(A_1'\). For \(A_1' \in \text{IOTSi}(\{i\}, \Sigma)\), the transition relation of \(A_1'\) is that of NFA \(A_1\) to which \(i\)-labeled self-loops are added at the every single state to make it input enabled.

Note that the above IOLTSs are constructed linearly in the size of the transition relation of \(A_1\) and \(A_2\). Using similar lines of reasoning in the proof of Theorem 5.2, we prove the above thesis by showing that \(L(A_1) \subseteq L(A_2)\) if and only if \(A_1' \text{ioco}^{a,b}_{\text{traces}(s)} A_2'\) with \(a, b \in \{\Pi, \omega\}\) and \(F = \text{ICtraces}(\bar{s}_2)\).

The above theorem shows that restricting the set of observation from suspension traces to internal-choice traces does not have any impact on the complexity of checking implementation relation \(\text{ioco}^{a,b}_{\text{traces}(s)}\) for \(a, b \in \{\Pi, \omega\}\) and \(F \subseteq \text{straces}(s)\) for a given specification \(s\), in general. In the rest of this section, we show that aforementioned relation can be decided in polynomial time for deterministic specifications. To this end, we first tailor the definition of the coinductive \(\text{ioco}\) relation \(R\) (Definition 5.4) such that \(R\) gets a witness for \(\text{ioco}^{a,b}_{\text{traces}(s)}\) relation where \(a, b \in \{\Pi, \omega\}\) and \(F \subseteq \text{ICtraces}(s)\) for the deterministic specification \(s\). Afterwards, we modify Algorithm 5.3.1, reflecting the change that we have made in Definition 5.4.

**Definition 5.19** (Coinductive \(\text{ioco}^{a,b}_{\text{traces}(s)}\)). Let deterministic IOLTS \(\langle S, I, U, \rightarrow_s, s \rangle\) be a specification, and let IOTS \(\langle Q, I, U, \rightarrow, \bar{r} \rangle\) be an implementation. A binary relation \(R \subseteq Q \times S\) is called a coinductive \(\text{ioco}^{a,b}_{\text{traces}(s)}\) relation from \(\bar{r}\) to \(s\) when \((\bar{r}, s) \in R\) and for each \((q, p) \in R\), then

- (Input simulation) if \(\delta(p)\) and \(p \xrightarrow{s} p'\), for \(c \in I\), then \((q \quad \text{after} \quad c) \neq \emptyset\) and for all \(q' \in q \quad \text{after} \quad c\), we have \((q', p') \in R\)

100
5.4. Conformance Checking for Internal-Choice Traces

• (Output simulation) if \( q \xrightarrow{c} q' \), and \( c \in U \cup \{\delta\} \), then \( c \in \text{out}(p) \) and for all \( p' \in p \) after \( c \), we have \( (q',p') \in R \).

We write \( r \preceq_{\text{ICtraces}(s)} s \), when there exists a coinductive \( \text{ioco}^{a,b} \) relation \( r \) to \( s \).

In the theorem below, we establish a connection between a coinductive \( \text{ioco}^{a,b} \) relation \( R \) and the implementation relation \( \text{ioco}^{a,b} \) with \( a, b \in \{\land, \lor\} \).

**Theorem 5.20.** For deterministic specifications, coinductive \( \text{ioco}^{a,b} \) and \( \text{ioco}^{a,b} \) coincide where \( a, b \in \{\land, \lor\} \).

**Proof.** The proof of the above thesis is similar to the proof of Theorem 5.5. \( \square \)

**Example 5.21.** Consider the IOLTS’s \( \bar{s} \) and \( \bar{r} \) presented in Figure 5.4 on page 99. IOLTS \( \bar{s} \) is a specification of a vending machine which sells tea. After receiving a coin, it either delivers tea, refunds the coin or accepts the order of adding sugar (for free). IOLTS \( \bar{r} \) is a formal model of a possible implementation of this vending machine. Upon receiving a coin, the machine \( \bar{r} \) chooses non-deterministically between serving tea or refunding the coin. In both cases, the user needs to insert another coin if he/she asks for sugar.

We define the binary relation \( R = \{(\bar{r}, \bar{s}), (r_1, s_1), (r_2, s_1), (r_3, s_2), (r_4, s_2)\} \). Clearly, for all pairs of states \((q,p) \in R\), the two input and output simulation conditions presented in Definition 5.19 are satisfied. Thus, relation \( R \) is an \( \text{ioco}^{\land} \) -coinductive relation and it is also a witness for \( \bar{r} \) \( \text{ioco}^{\land} \bar{s} \) with \( F = \text{ICtraces}(\bar{s}) \). Notice that \( \bar{r} \) \( \text{ioco}^{\land} \bar{s} \), because for example \( \text{out}(\bar{r} \text{ after } \sigma) \notin \text{out}(\bar{s} \text{ after } \sigma) \) for \( \sigma = \text{coin sugar} \).

Theorem 5.20 provides us the facility to reduce the problem of checking \( \text{ioco}^{a,b} \) for a deterministic specification \( s \) to the satisfiability problem of a negative HORN formula. That reduction can be carried out in a similar way as that of the \( \text{ioco} \) relation, presented in Algorithm 5.3.1. By changing the loop condition on line (i) from \( a \in \text{init}(p) \cap I \) to \( a \in \text{init}(p) \cap I \land \delta(p) \), we can adjust Algorithm 5.3.1 for checking \( \text{ioco}^{a,b} \) relation between a deterministic specification \( s \) and a possibly non-deterministic implementation \( \bar{r} \). The correctness of the above reduction can be easily proven in similar lines of reasoning as in the proof of Theorem 5.13.

**Example 5.22.** Consider again the IOLTSs \( s \) and \( r \) presented in Figure 5.4. We know from Example 5.21 that \( \bar{r} \) \( \text{ioco}^{\land} \bar{s} \) with \( F = \text{ICtraces}(\bar{s}) \). The formula \( F \) generated by the modified version of Algorithm 5.3.1 mentioned above is the following:

\[
X_{\bar{r}s} \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \\
(\land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \\
(\land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \\
(\land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s}) \land (X_{\bar{r}s} \Rightarrow X_{\bar{r}s})
\]

Clearly, the constructed formula is satisfiable: assigning true to all literals is a satisfying assignment.

**Theorem 5.23.** Let \( \{S, I, U, \rightarrow_{s}, s\} \) be a deterministic IOLTS and let \( \{Q, I, U, \rightarrow_{r}, r\} \) be an arbitrary IOLTS over language \( L = I \cup U \). Deciding whether \( r \) \( \text{ioco}^{a,b} \) for \( a, b \in \{\land, \lor\} \) can be done in \( O(|L_{\delta}| \times |Q|^2) + O(|S| \times |Q| \times |L_{\delta}|) \).

**Proof.** The proof of the above thesis is identical to the proof of Theorem 5.15 if the modified version of Algorithm 5.3.1, mentioned above, is considered as the reduction algorithm. \( \square \)
5.5 Closing Remarks

In this chapter, we studied the complexity of checking input-output conformance (ioco). We proved that the problem of checking conformance is PSPACE-complete, in general. Then, we presented a coinductive definition of the ioco relation. Through a reduction of the ioco coinductive relation to NHORNSAT in the restricted setting of deterministic specifications, we presented a polynomial-time algorithm for checking the ioco relation in this setting. Such an algorithm cannot be developed for non-deterministic specifications, where a set of states is reachable after a trace.

Taking a similar approach in computing the complexity of the ioco relation, we calculating the complexity of the relation $ioco^{a,b}_F$ with $a, b \in \{\sqcap, \sqcup\}$, defined in Chapter 4, when $F$, the set of observation, is the set of internal-choices traces of a given specification.

The coinductive ioco relation, introduced in Section 5.2, closely resembles alternating refinement of Interface Automata. So, that similarity encourages us to investigate the application of our algorithm in Section 5.3.2 to checking alternating simulation. Currently, the best known algorithm for this purpose is proposed in [CCK12], which is a game-based algorithm for deterministic models. The solution provided in this paper may offer an alternative for the existing algorithms for checking alternating refinement relation, but it must be checked to see whether the runtime complexity of the resulting algorithm is comparable to that of existing algorithms.

Marie-Claude Gaudel in [Gau95] introduced a test selection hypothesis that is known as regularity hypothesis in testing based on algebraic specifications. In this case, if all test cases up to a certain level of complexity (data size) are passed, then every test case after that level will be passed as well. We can translate the regularity hypothesis into the setting of the ioco testing theory by identifying a limit on the length of test cases. In this regard, if all test cases up to a certain length (henceforth referred to as the length of regularity) are passed, then the hypothesis is that any longer test case will also be passed. We say that a subset of ioco test cases is regular if and only if it confirms the regularity hypothesis. In this regard, testing the minimum set of regular test cases, which is indeed finite, is sufficient to judge about the conformance of the IUT with its specification. Our study in this chapter gives us an insight to be able to identify the minimum length of regularity that is needed by a regular set of test cases. In this regard, we say a pair of states $(q, p)$, where $q$ and $p$ are, respectively, two states in the IUT and the specification, is synchronously reachable whenever there is a trace from the initial states to stats $q$ and $p$, respectively. Subsequently, the distance of a synchronously reachable pair is defined as the shortest trace to that pair from the initial states. From the coinductive ioco relation in Section 5.2 together with our reduction algorithm in Section 5.3, we obtain that the minimum length of regularity has to guarantee that the farthest synchronously reachable pair is visited. Identifying the longest distance between synchronously reachable pairs is only possible by making stronger assumptions about the IUT than the testing hypothesis [Hie+09].
Chapter 6

Asynchronous Input-Output
Conformance Testing

Due to the ubiquitous presence of distributed systems (ranging from distributed embedded systems to the Internet), it becomes increasingly important to establish rigorous model-based testing techniques with an asynchronous model of communication in mind. This fact has been noted by the pioneering pieces of work in the area of formal conformance testing, e.g., see [Tre92, Chapter 5], [TV92] and [Ver+93], and has been extensively addressed by several researchers in this field ever since [Jar+99; PYH03; WW09; Wei09; SP10; SP11; Dav+13].

Unlike when using synchronous communication, because of lags in asynchronous communication channels, the tester is not able to observe the exact time of the consumption of an input nor the production time of an output by the IUT. These communication delays may cause that the behavior of the IUT observed by the tester differs from the actual behavior of the IUT. Therefore, the tester may arrive at a fail verdict while there is no actual fault in the IUT; this phenomenon is henceforth referred to as a false fail verdict.

We stumbled upon this problem in our attempt to apply the ioco testing theory [Tre96c; Tre08] to the EFT switch from Chapter 3. In that experiment, the tester communicates (using an adapter) via a network with the IUT. The communication is inherently asynchronous and hence subtleties concerning asynchronous testing arise naturally in our context. Figure 6.1 shows a simplified specification of the purchase transaction in the switch, in which these subtleties appear. In this figure, the switch sends a purchase request to the core banking system and either receives a response, or after an internal step (e.g., an internal time-out, denoted by \( \tau \)) sends a reversal request to the POS. In the synchronous setting, after sending a purchase request and receiving a response, observing a reversal request will lead to the fail verdict. This is justified by the fact that receiving a response should force the system to take the topmost transition at the moment of choice in the specification depicted in Figure 6.1. However, in the asynchronous setting, a response is put on a channel and is yet to be communicated to the IUT (see Section 2.1.1). It is unclear to the remote observer (the tester) when the response is actually consumed by the IUT. Hence, even when a response is sent to the system the observer should still expect to receive a reversal request.
Chapter 6. Asynchronous Input-Output Conformance Testing

Figure 6.1: IOLTS \( \delta \) is the simplified specification of the purchase transaction with the inputs \( \{\text{p}_{rs}\} \) and the outputs \( \{\text{p}_{rq}, \text{r}_{rq}\} \)

The problems encountered in our practical case study have been observed by other researchers as well. It is well-known [WW09; SP10; SP14] that not all systems are amenable to asynchronous testing since they may feature phenomena (e.g., a choice between accepting input and generating output) that cannot be reliably observed in the asynchronous setting (e.g., due to unknown delays). The first intuitive solution to this problem, is to take the test execution context into account in testing [Tre92], i.e., in this setting, the queue context [Ver+93]. As we reported on our experience in Chapter 3, this solution suffers from the infamous problem of state space explosion, which happens because of composing queue models with a given specification. Besides that problem, some of the test cases that are derived from the composition of the given specification with the queues do not examine an actual behavior of the system under test. For example, according the purchase scenario depicted in Figure 6.1, the switch naturally does not receive a purchase response unless it sends out a purchase request beforehand. By composing that specification with queues, a tester can check trace \( \text{p}_{rs} \text{p}_{rq} \); a situation in which the system under test receives a response before it even requests it. That trace does not represent any meaningful behavior of the system. Therefore, checking such traces in testing does not necessarily lead to detecting a real defect in the IUT, which is the ultimate goal of testing.

Many studies were conducted on conformance testing in the asynchronous setting to sidestep the above-mentioned problems in test case derivation caused by composing the specification with queues. To this end and to make sure that test cases generated from the specification -not its composition with queues- can test the IUT using asynchronous interactions and reach meaningful verdicts, either the class of implementations, the class of specifications, or the test cases (or a combination thereof) has to be adapted. Such adaptations, however, may modify the testing power of underlying implementation relations in the asynchronous setting in comparison with that in the synchronous setting. An overview of some of those works are given below.

In [Wei09, Chapter8] and [WW09], to avoid the problems of the ioco testing theory in the queue context, both the class of implementations and the class of specifications have been confined to internal-choice IOLTSs. Furthermore, the test case generation algorithm is adapted to generate internal-choice test cases. Then, it is argued (with a proof sketch) that in this setting, the verdict obtained through asynchronous interaction with the system coincides with the verdict (using the same set of restricted test cases) in the synchronous setting. In Chapter 4, we proved that the implementation relation implied by internal-choice test cases coincide with the ioco relation for internal-choice IOLTS's in the synchronous settings. In this chapter, we, therefore, investigate whether using internal choice test cases leads to a solution for the use of the ioco relation in the asynchronous setting. To this end, we give a full proof of the main claim of [Wei09] in Section 6.1 and
6.1 Adapting the ioco Relation to Asynchronous Setting

report a slight adjustment to that, without which a counter-example is shown to violate the property. However, we show in Section 6.2 that our results in Chapter 4 are not valid in the asynchronous context; the testing power of the implementation relation implied by internal-choice test cases are different from that of the ioco relation in the queue context.

In [SP10] the test case generation algorithm has been adapted to the asynchronous setting. A method is presented for generating test cases from a synchronous specification that are sound for an asynchronous implementation. The main idea is to saturate a test case with observation delays caused by the asynchronous interactions. In this chapter, we adopt a restriction imposed on the class of implementations inspired by [SP10, Theorem 1] (dating back to [Tre92]) and prove that in the setting of ioco testing, this is sufficient for using synchronous test case for an asynchronous implementation.

In [PYH03] the asynchronous test framework is extended to the setting where separate test processes can observe input and output events and the relative distinguishing power of these settings are compared. Although this framework may be natural in practice, we avoid following the framework of [PYH03] since our ultimate goal is to compare asynchronous testing with the standard ioco framework and the framework of [PYH03] is notationally very different. For the same reason, we do not consider the approach of [Jar+99], which uses a stamping mechanism attached to the IUT, thus observing the actual behavior of the IUT before being distorted by the queues.

In this chapter, we formulate and prove several theorems which characterize when the ioco relation in the queue context can be used once test cases are derived from the specification (not its composition with queues). These theorems define when the synchronous test cases are sufficient for checking all aspects of conformance that are observable by asynchronous interactions with the implementation under test. To summarize, in the present chapter, we first identify a subclass of IOTSs as behavioral models of implementations for which test verdicts of ioco test-cases derived from a given specification in the asynchronous setting are the same as the verdicts obtained from the same test cases in synchronous testing. Afterwards, we restrict the class of specifications to a subclass of IOLTSs such that standard ioco-test cases are sound in the asynchronous communication.

Structure of the chapter. In Section 6.1, we give a full proof of the main result of [Wei09, Chapter 8] and [WW09] (and report on a slight imprecision). Then, in Section 6.2, we re-formulate the same results in the pure ioco setting and show that our constraints precisely characterize the implementations for which asynchronous testing can be reduced to synchronous testing. Subsequently, in Section 6.3, we introduce a subset of IOLTSs, whose ioco test cases always yield meaningful verdicts in the asynchronous setting. Finally, the chapter is concluded in Section 6.4.

6.1 Adapting the ioco Relation to Asynchronous Setting

In order to use the ioco testing theory in the asynchronous setting in [Wei09] and [WW09] the class of implementations, specifications, and test cases are restricted to the internal-choice class (see Section 2.1.2 and Section 2.2). Then, it is argued (with a proof sketch) that in this setting, the verdict obtained through asynchronous interaction with the system coincides with the verdict (using the same set of internal-choice test cases) in the
synchronous setting. In this section, we re-visit the approach of [Wei09] and [WW09], give a full proof of their main result and point out a slight imprecision in it.

In [Wei09; SP10], it is argued that providing inputs to the IUT only after observing quiescence (i.e., when the IUT is in a stable state), eliminates the distortions in observable behavior, introduced by communicating with the IUT using queues, see Section 2.1.1. Hence, a subset of synchronous test cases, namely internal-choice test cases which only provide an input after observing quiescence, are safe for testing asynchronous systems. This is summarized in the following claim from [Wei09; WW09] (and paraphrased in [SP10]):

Claim 6.1 (Theorem 1 in [Wei09]). Let $\bar{s}$ be an arbitrary $IOTS^\tau$, and let $\bar{t}$ be a $TTS^\tau$. Then $\bar{s}$ passes $\bar{t}$ iff $Q(\bar{s})$ passes $\bar{t}$.

In [SP10], the claim is taken for granted, and, unfortunately, in [WW09; Wei09] only a proof sketch is provided for the above claim; the proof sketch is rather informal and leaves some room for interpretation, as illustrated by the following excerpt [WW09]:

"...An implementation guarantees that it will not send any output before receiving an input after quiescence is observed..."

As it turns out, the above result does not hold in its full generality, as illustrated by the following example.

Example 6.2. Consider the internal-choice test case with initial state $t_0$ in Figure 6.2. Consider the implementation modeled by the $IOTS^\tau$ depicted in Figure 6.2. Clearly, we find that $o_0$ passes $t_0$; however, in the asynchronous setting, $Q(o_0)$ passes $t_0$ does not hold. This is due to the divergence in the implementation due to state $o_1$, which gives rise to an observation of quiescence in the queue context, but not so in the synchronous setting. Note that the external observer in the queue context cannot tell whether a system is involved in some internal transitions or it is standstill.

The claim does hold for non-divergent internal-choice implementations. Note that divergence is traditionally also excluded from testing theories such as ioco. In this sense,
6.1. Adapting the \textit{ioco} Relation to Asynchronous Setting

Figure 6.3: A candidate implementation of and an internal-choice test case designed for the purchase scenario depicted in Figure 6.1 with the inputs \{p_{rs}\} and the outputs \{p_{rq}, r_{rq}\} assuming non-divergence is not a severe restriction. Apart from the following theorem, we tacitly assume in all our formal results to follow that the implementation IOLTSs are non-divergent.

**Theorem 6.3.** Let \(\langle S, I, U, \rightarrow_s, \bar{s}\rangle\) be an arbitrary IOTS\(^{\uparrow}\) and let \(\langle T, U \cup \{\theta\}, I, \rightarrow_t, \bar{t}\rangle\) be a TTS\(^{\uparrow}\). If \(\bar{s}\) is non-divergent, then \(\bar{s}\) passes \(\bar{t}\) iff \(Q(\bar{s})\) passes \(\bar{t}\).

Given the pervasiveness of the original (non-)theorem, a formal correctness proof of our corrections to this theorem (i.e., our Theorem 6.3) is highly desirable. In the remainder of this section, we therefore give the main ingredients for establishing a full proof for Theorem 6.3.

**Example 6.4.** Consider internal-choice test case \(\bar{t}\) and IOTS \(\bar{r}\) depicted in Figure 6.3. IOTS \(\bar{s}\) is given as a candidate implementation for the purchase scenario specified by IOLTS \(\bar{s}\) in Figure 6.1. As mentioned above, when an ordinary test case asynchronously interacts with IOTS \(\bar{r}\), a distortion in the sequence of inputs and outputs happens in the test execution because of the queues; after executing sequence \(p_{rq}\), while input \(p_{rs}\) is provided by the tester, output \(p_{rq}\) can be emitted by \(\bar{r}\). Thus, the tester can observe output \(p_{rq}\) after providing input \(p_{rs}\) in the queue context, which is not allowed. Now, consider the test execution of TTS\(^{\uparrow}\) \(\bar{t}\) against IOTS \(\bar{r}\). We observe that state \(r'\) that is reachable by output \(p_{rq}\) from \(\bar{r}\) is not quiescent. Therefore, test case \(\bar{t}\) after executing sequence \(p_{rq}\) cannot provide any input even when the test execution is asynchronously carried out, i.e., \(Q(r') \not\xrightarrow{p_{rq}} t'\). Test case \(\bar{t}\) thus after sequence \(p_{rq}\) always waits for outputs of \(\bar{r}\), i.e., output \(r_{rq}\) in this case. By observing quiescence before providing any input, test case \(\bar{t}\) prevents distortions that are caused by queues in the test execution.

As illustrated in the above example, by using internal-choice test cases, we have some kind of hand-shaking protocol between testers and the IUT for synchronization even in the asynchronous setting. The Observation of quiescence before any input action is taken as the synchronization point between testers and the IUT. In the remainder of this section, we show that the test runs obtained from executing internal-choice test cases on the IUT which behaves as an IOTS\(^{\uparrow}\) is independent of the test execution context; test executions of
an internal-choice test case in the asynchronous setting which is modeled as FIFO queues are the same as its text executions in the synchronous setting. We know from Property 2.32 that every test execution in the synchronous setting can be executed in the queue context as well. We, therefore, study how the test runs obtained in the queue context can be related to the test executions in the synchronous setting when the behavioral models of test cases and the IUT are confined to IOLTS’s. Before establishing such a link, we first formalize some characteristics of the test executions of an internal-choice test case on the IUT behaving as an IOTS while interactions between them are carried out via queues.

Note that due to the lags in the asynchronous communication channels, the tester cannot distinguish whether the IUT is actively engaged in some internal transitions or is stable. We, therefore, use the notion of weak quiescence in the queue context, see Section 2.1.2. We proceed by establishing a formal correspondence between observations of quiescence in the synchronous setting and observations of weak quiescence in the asynchronous setting.

**Proposition 6.5.** Let \( \langle s, I, U, \rightarrow, \bar{s} \rangle \) be an IOTS. Let \( s \in S \) be an arbitrary state. Then \( \delta_q(Q(s)) \) implies \( \delta(s') \) for some \( s' \in S \) satisfying \( s \Rightarrow s' \).

**Proof.** Assume, towards a contradiction, that for all \( s' \in S \) such that \( s \Rightarrow s' \), it does not hold \( \delta(s') \). Take the \( s' \) with the largest empty trace (by counting the numbers of \( \tau \)-labeled transitions). Such \( s' \) must exist since otherwise, there must be a loop of \( \tau \)-labeled transition which contradicts the assumption that \( s \) does not diverge. Since \( s' \) is not quiescent, according to Definition 2.12, there exists an \( x \in U \) such that \( s' \xrightarrow{\tau} \). Hence, there must exist an \( s'' \in S \) such that \( s' \xrightarrow{\tau} s'' \). It follows from Proposition 2.8 and deduction rule \( l2r \) in Definition 2.11 that \( Q(s) \Rightarrow \{ x \in s'' \} \) and since the output queue is non-empty we can apply the deduction rule \( A2r \) in Definition 2.11 on the target state and obtain \( \{ x \in s'' \} \xrightarrow{\tau} Q(s'') \). Combining the two transitions, we obtain that \( Q(s) \Rightarrow Q(s'') \). From the latter transition we can conclude that \( Q(s) \) is not quiescent which is contradictory to the statement.

The above proposition guarantees that all stimuli provided by an internal-choice test case are accepted by implementations that behave as some IOTS, even when we adopt the asynchronous communication scheme between testers and the implementation. Therefore, any test execution in the asynchronous setting can lead eventually to a state in which both communication queues are empty. Before we give the formal proposition of the above claim, we first demonstrate through the following lemmas how the use of internal-choice models imposes restrictions on the test execution in the queue context ensuring that no distortion in the observable behavior of the IUT happens. The lemma below states that only at weakly quiescent states the input queue can grow.

**Lemma 6.6.** Let \( \langle s, I, U, \rightarrow, \bar{s} \rangle \) be an IOTS, and let \( \langle T, U \cup \{ \varnothing \}, I, \rightarrow, \bar{t} \rangle \) be a TTS. Let \( s, s' \in S \), and \( t, t' \in T \) be arbitrary states, and let \( \sigma_u \in U^* \) and \( \sigma_i \in I^* \) and \( a \in I \). If \( t \xrightarrow{s, s' \in \sigma_i} a \xrightarrow{t} t' \xrightarrow{s, s' \in \sigma_i} \), then \( \delta_q(q(s, s' \in \sigma_i)) \).

**Sketch of the proof.** We obtain the correctness of the above thesis from observing that internal-choice test case \( \bar{t} \) only provides an input immediately after if it has observed
Following the above lemma, observing (weak) quiescence is essentially a synchronization point between a TTS and the IUT. Consequently, we find that during the execution of an internal choice test case, the input and output queues cannot be non-empty simultaneously. This is formalized by the lemma below.

**Lemma 6.7.** Let \( \langle S, I, U, \rightarrow, \bar{s}, \bar{t} \rangle \) be an IOTS, and let \( \langle T, U \cup \{\theta\}, I, \rightarrow, \bar{t} \rangle \) be a TTS. Let language \( L' = U \cup I \cup \{\theta\} \), and let \( s, s' \in S, t, t' \in T \) be arbitrary states. There is no trace \( \sigma_u \in L' \) such that \( t \parallel Q(s) \Rightarrow t' \parallel [\sigma_u \vDash s' \vDash \sigma_i] \) and the input and output queues are both non-empty at the same time (\( \sigma_i \neq \epsilon \land \sigma_u \neq \epsilon \)).

**Sketch of the proof.** Assume, towards a contradiction, that the following two statements hold:

1. \( t \parallel Q(s) \Rightarrow t' \parallel [\sigma_u \vDash s' \vDash \sigma_i] \)
2. \( \sigma_i \neq \epsilon \land \sigma_u \neq \epsilon \)

Since both \( \sigma_i \) and \( \sigma_u \) are non-empty, there must exist some largest prefix \( \sigma' \) of \( \sigma \) during which the two queues are never simultaneously non-empty. By observing a single action after \( \sigma' \), both queues become non-empty for the first time. Hence, there exists \( \sigma', \sigma'' \in L' \) as a prefix and suffix of \( \sigma \), respectively, and \( y \in L' \cup \{\tau\} \) such that \( \sigma = \sigma'y\sigma'' \) and one of the following transitions is executed:

1. \( t \parallel Q(s) \Rightarrow t_1 \parallel [\epsilon \vDash s_1 \vDash \sigma_i] \Rightarrow t_2 \parallel [x \vDash s_2 \vDash \sigma_i] \Rightarrow t' \parallel [\sigma_u \vDash s' \vDash \sigma_i] \) with \( t_1 \in T \) and \( s_1 \in S \) for some \( \sigma_i' \in I^* \) and \( x \in U \).
2. \( t \parallel Q(s) \Rightarrow t_1 \parallel [\sigma_u' \vDash s_1 \vDash \epsilon] \Rightarrow t_2 \parallel [\sigma_u' \vDash s_2 \vDash x] \Rightarrow t' \parallel [\sigma_u \vDash s' \vDash \sigma_i] \) with \( t_1 \in T \) and \( s_1 \in S \) for some \( \sigma_u' \in U^* \) and \( x \in I \).

Note that the transition \( y \), which only increases the size of one of the two queues, is taken only under either rules A1 or I2 in Definition 2.11. In both of the above cases, we can easily verify that one of the input and output queues must be empty, otherwise the internal-choice test case \( t \) provides an input while \( Q(s) \) is not quiescent.

Finally, the lemma given below states that in a queue context, implementations that have a non-empty input queue are weakly quiescent. The correctness of the lemma follows from the two preceding lemmata.

**Lemma 6.8.** Let \( \langle S, I, U, \rightarrow, \bar{s}, \bar{t} \rangle \) be an IOTS, and let \( \langle T, U \cup \{\theta\}, I, \rightarrow, \bar{t} \rangle \) be a TTS. Let language \( L' = I \cup U \cup \{\theta\} \), and let \( s, s' \in S, t, t' \in T \) be arbitrary states, \( \sigma \in L'^* \), \( \sigma_i \in I^* \) and \( \sigma_u \in U^* \). If \( t \parallel Q(s) \Rightarrow t' \parallel [\sigma_u \vDash s' \vDash \sigma_i] \) and \( \sigma_i \neq \epsilon \) then \( \delta_q(s') \) and \( \sigma_u = \epsilon \).

**Proof.** Following lemma 6.7, we have that \( \sigma_u = \epsilon \). The correctness of the second part of the above statement directly follows from that internal-choice test case \( \bar{t} \) only provides an input immediately after it has observed quiescence.
After characterizing the test execution of a TTS in a queue context, we are now in a position to formally establish the proposition below which states that both queues can eventually be empty after running an internal-choice test cases in the asynchronous setting.

**Proposition 6.9.** Let \( \langle S, I, U, \rightarrow_S, s, \bar{s} \rangle \) be an IOTS, and let \( \langle T, U \cup \{\theta\}, I, \rightarrow_T, \bar{t} \rangle \) be a TTS. Let language \( L' = U \cup I \cup \{\theta\} \). Assume arbitrary states \( t' \in T \) and \( s' \in S \), and an arbitrary test run \( \sigma \in L^* \). Then for all \( \sigma_i \in I^* \) and \( \sigma_u \in U^* \):

\[
\bar{t} \mathcal{Q}(\bar{s}) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(s') \text{ implies } \exists s'' \in S \cdot \bar{t} \mathcal{Q}(\bar{s}) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(s'')
\]

**Proof.** We distinguish four cases based on the status of the input and the output queues.

1. **Case \( \sigma_i = \epsilon \), \( \sigma_u = \epsilon \).** By assuming \( s'' = s' \), the statement holds.

2. **Case \( \sigma_i \neq \epsilon \), \( \sigma_u \neq \epsilon \).** According to Lemma 6.7, no trace leads to this situation.

3. **Case \( \sigma_i \neq \epsilon \), \( \sigma_u = \epsilon \).** We prove this case by an induction on the length of \( \sigma_i \).

   - **Base case.** Since \( \sigma_i \neq \epsilon \), for the base case, the smallest possible length of \( \sigma_i \) is one. Thus there must be an \( x \in I \) such that \( \sigma_i = x \). From Lemma 6.8, we know that \( \forall y \in U, y \notin \text{Sinit}(s') \) and since \( s' \) does not diverge, it must reach eventually a state \( i \in S \) which performs a transition other than an internal one, hence the only possible choice is an input transition. From Definition 2.19 we know that \( \delta(i) \) and state \( i \) is input enabled as well. Thus \( \exists i' \in S \cdot i \xrightarrow{x} i' \). Due to the subsequent application of deduction rules of II, I3 in Definition 2.11 and RI in Definition 2.29, transition \( t' \mathcal{Q}(\bar{s}) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(i') \) is possible. By assuming \( s'' = s' \) and combining the latter transition and the assumption, we have \( \bar{t} \mathcal{Q}(\bar{s}) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(s'') \) which was to be shown.

   - **Induction step.** For the induction step, we assume that the statement holds for all non-empty input queues of length \( n - 1 \) or less and length \( n \) for \( \sigma_i \). It follows from \( \sigma_i \neq \epsilon \) that there exists an \( a \in I \), \( \sigma_i' \in I^* \), \( \sigma_i'' \in L'^\ast \) and \( i' \in S \) and \( t_p \in T \) such that \( \sigma_i = \sigma_i'a \) and \( t \mathcal{Q}(s) \overset{\sigma}{\Rightarrow} t_p \mathcal{Q}(s') \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(s'') \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(i') \). It follows from the induction hypothesis that \( \exists i' \in S \cdot i' \mathcal{Q}(s') \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(i') \). Due to the application of deduction rule R2 in Definition 2.29 and AI in Definition 2.11, we have \( t_p \mathcal{Q}(i) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(i') \). It follows from the induction basis that \( \exists s'' \in S \cdot t_p \mathcal{Q}(i) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(s'') \). Combining both transitions leads to \( \exists s'' \in S \cdot \bar{t} \mathcal{Q}(\bar{s}) \overset{\sigma}{\Rightarrow} t' \mathcal{Q}(s'') \) which was to be shown.

4. **Case \( \sigma_i = \epsilon \), \( \sigma_u \neq \epsilon \).** We prove this case by an induction on the length of \( \sigma_u \).

   - **Base case.** Since \( \sigma_u \neq \epsilon \), for the induction basis, the smallest possible length of \( \sigma_u \) is one. Thus, we assume, for the induction basis, that there exists an \( x \in U \) such that \( \sigma_u = x \). The only possible transition that can fill the output queue is due to the application of deduction rule I2 in Definition 2.11. Hence, there must exist some \( s'', q'' \in S \) such that \( \sigma_u \mathcal{Q}(s'' \ll \sigma_u) \overset{\tau}{\Rightarrow} \sigma_u \mathcal{Q}(q'' \ll \sigma_u) \overset{\epsilon}{\Rightarrow} \).
Corollary 6.10. Let \( \langle s, I, U, \rightarrow_s, \bar{s} \rangle \) be an IOTS\( ^\bar{\gamma} \), and let \( \langle T, U \cup \{ \theta \}, I, \rightarrow_t, \bar{\theta} \rangle \) be a TTS\( ^\gamma \). Let language \( L' = I \cup U \cup \{ \theta \} \). Assume arbitrary states \( t' \in T \) and \( s', \bar{s}' \in S \), and an arbitrary test run \( \sigma \in L'^\epsilon \) and \( x \in L' \). Then \( t \| Q(s) \stackrel{\sigma}{\Rightarrow} t' \| Q(s') \) implies \( \exists t'' \in T, s'' \in S \cdot t \| Q(s) \stackrel{\sigma}{\Rightarrow} t'' \| Q(s'') \) which was to be shown.

\[\Box\]

As a consequence of the above proposition, we find the following corollary. It states that each asynchronous test execution can be chopped into individual observations such that before and after each observation the communication queues are empty.

**Corollary 6.11.** Let \( \langle S, I, U, \rightarrow_s, \bar{s} \rangle \) be an IOTS\( ^\bar{\gamma} \), and let \( \langle T, U \cup \{ \theta \}, I, \rightarrow_t, \bar{\theta} \rangle \) be a TTS\( ^\gamma \). Let language \( L' = I \cup U \cup \{ \theta \} \). Assume arbitrary states \( t' \in T \) and \( s', \bar{s}' \in S \), and an arbitrary test run \( \sigma \in L'^\epsilon \) and \( x \in L' \). Then \( t \| Q(s) \stackrel{\sigma}{\Rightarrow} t' \| Q(s') \) implies \( \exists t'' \in T, s'' \in S \cdot t \| Q(s) \stackrel{\sigma}{\Rightarrow} t'' \| Q(s'') \) which was to be shown.

\[\Box\]
cases and the IUT are adopted to IOLTS\(^\cap\)'s. Moreover, it formalizes the relation between reachable states after a test run in the synchronous settings with the one reachable in the asynchronous setting. This lemma is basic to the correctness of our main results in this section.

**Lemma 6.11.** Let \(\langle S, I, U, \rightarrow, \bar{s}, \bar{I}\rangle\) be an IOLTS\(^\cap\), and let \(\langle T, U \cup \{\theta\}, I, \rightarrow, \bar{t}, \bar{I}\rangle\) be a TTS\(^\cap\). Let language \(L' = I \cup U \cup \{\theta\}\), and let \(s' \in S\) and \(t' \in T\) be arbitrary states. Then, for all \(\sigma \in L'^\star\), such that \(\bar{t} \rhd Q(s')\), there is a non-empty set \(\mathbb{S} \subseteq \{s'' \in S \mid s' \xrightarrow{\sigma} s''\}\) such that

1. \(\{s'' \in S \mid \delta(s'') \wedge s' \xrightarrow{\sigma} s''\} \subseteq \mathbb{S}\) if \(\exists \sigma' \in L'^\star \cdot \sigma = \sigma' \theta\)
2. \(s' \in \mathbb{S}\) if \(\exists \sigma' \in L'^\star \cdot \sigma = \sigma' \theta\)
3. \(\forall s'' \in \mathbb{S} \cdot \bar{t} \rhd s''\).

**Proof.** We prove this lemma by induction on the length of \(\sigma \in L'^\star\).

- **Base case.** We assume that the length of \(\sigma\) is 0, i.e., \(\sigma = \epsilon\). Assume that \(\bar{t} \rhd Q(s')\). We find from Definition 2.31 that \(Q(s) \xrightarrow{\epsilon} s\ Q(s')\). By Definition 2.11, an internal transition is performed only under rule I1. By using an induction on the number of \(\tau\)-labeled transitions leading to \(\Rightarrow_s\), we obtain that \(Q(s) \Rightarrow_s Q(s')\) results in \(s \xrightarrow{\epsilon} s'\). Consider \(\mathbb{S} = \{s'' \mid s' \xrightarrow{\epsilon} s''\}\). Let \(s'' \in \mathbb{S}\) be an arbitrary state. By Definition 2.29, we find that \(\bar{t} \rhd s\ \Rightarrow \bar{t} \rhd s'\) and \(\bar{t} \rhd s' \Rightarrow \bar{t} \rhd s''\); by transitivity, we have the desired \(\bar{t} \rhd s \Rightarrow \bar{t} \rhd s''\). It is also clear that \(s' \in \mathbb{S}\). We thus find that \(\mathbb{S}\) meets the desired conditions.

- **Induction step.** We assume that the statement holds for all \(\sigma'\) of length at most \(n - 1\). Suppose that the length of \(\sigma\) is \(n\). Assume that \(\bar{t} \rhd Q(s)\). By Corollary 6.10, there is some \(s_{n-1} \in S\), a \(t_{n-1} \in T\) and \(\sigma_{n-1} \in L'^\star\) and \(x \in L'\), such that \(\sigma = \sigma_{n-1} \cdot x\) and \(\bar{t} \rhd Q(s) \xrightarrow{\sigma_{n-1}} t_{n-1} \rhd Q(s_{n-1}) \xrightarrow{x} t' \rhd Q(s')\). By induction, there must be a set \(\mathbb{S}_{n-1} \subseteq \{s'' \in S \mid s_{n-1} \xrightarrow{\sigma_{n-1}} s''\}\), such that

1. \(\{s'' \in S \mid \delta(s'') \wedge s_{n-1} \xrightarrow{\sigma_{n-1}} s''\} \subseteq \mathbb{S}_{n-1}\) if \(\exists \sigma' \in L'^\star \cdot \sigma = \sigma' \theta\)
2. \(s_{n-1} \in \mathbb{S}_{n-1}\) if \(\exists \sigma' \in L'^\star \cdot \sigma = \sigma' \theta\)
3. \(\forall s'' \in \mathbb{S}_{n-1} \cdot \bar{t} \rhd s''\).

We next distinguish three cases: \(x = \theta, x \in I, x \in U\).

1. We assume that \(x = \theta\). We thus find that \(t_{n-1} \rhd Q(s_{n-1}) \Rightarrow t_n \rhd Q(s')\). As a result of Corollary 6.10, we have \(\delta_Q(s')\). We then find as a result of Lemma 6.5, there must be some state \(s'' \in S\) such that \(s_{n-1} \xrightarrow{\sigma_{n-1}} s' \xrightarrow{\epsilon} s''\) and \(\delta(s'')\). Consider the set \(\mathbb{S}_n = \{s'' \in S \mid \delta(s'') \wedge s' \xrightarrow{\epsilon} s''\}\). Let \(s''\) be an arbitrary state in \(\mathbb{S}_n\). We distinguish between cases \(s_{n-1} \notin \mathbb{S}_{n-1}\) and \(s_{n-1} \in \mathbb{S}_{n-1}\). In the case, \(s_{n-1} \notin \mathbb{S}_{n-1}\), we know from the construction of \(\mathbb{S}_{n-1}\) that \(s'' \in \mathbb{S}_{n-1}\) and \(s'' \xrightarrow{\epsilon} s''\) always holds. In the case \(s_{n-1} \in \mathbb{S}_{n-1}\), we
have that \( s_{n-1} \xrightarrow{\epsilon} s' \xrightarrow{\epsilon} s'' \). We thus find that \( \forall s'' \in S_n \exists s \in S_{n-1} \cdot \bar{t} \xrightarrow{\sigma_{n-1}} t_{n-1} \{s\} \Rightarrow t_{n-1} \{s'' \theta \} \Rightarrow t' \{s''\} \).

Thus \( S_n \) has the desired requirement that \( \bar{t} \xrightarrow{\sigma_{n-1}} t' \{s''\} \) for all \( s'' \in S_n \). Also, \( \{s'' \in S \mid \delta(s'') \land s' \xrightarrow{\epsilon} s''\} \subseteq S_n \) is concluded from construction of \( S_n \). Hence, \( S_n \) satisfies all desired conditions.

2. We assume that \( x \in I \). By Property 4.37, we find that the last step in \( \sigma_{n-1} \) must be \( \theta \). It follows from corollary 6.10 that \( Q(s_{n-1}) \) is weakly quiescent and consequently \( \delta_Q(s_{n-1}) \). By induction we have that \( \{s'' \in S \mid \delta(s'') \land s' \xrightarrow{\epsilon} s''\} \subseteq S_{n-1} \). Consider the set \( S_n = \{s'' \in S \mid s' \xrightarrow{\epsilon} s''\}; \) transition \( t_{n-1} \}Q(s_{n-1}) \Rightarrow t' \}Q(s') \) implies that \( s_{n-1} \xrightarrow{\epsilon} s' \). By Lemma 6.5 and Definition 2.19, we know that \( \exists s \in S \) such that \( s_{n-1} \xrightarrow{\epsilon} s' \). From construction of \( S_n \), we know that \( s \) is in \( S_n \). We thus have \( \forall s'' \in S_n \exists s \in S_{n-1} \cdot \bar{t} \xrightarrow{\sigma_{n-1}} t_{n-1} \{s\} \Rightarrow t' \{s''\} \).

It is clear form construction of \( S_n \) that \( s' \in S_n \) as the required condition that \( s' \in S_n \) if the last step of \( \sigma \) is not \( \theta \)-labeled transition. We thus find that \( S_n \) fulfills all of the desired requirements.

3. We assume that \( x \in U \). Analogous to the previous case.

\( \square \)

Through the above propositions, we thus far showed that test executions of an \( \text{TTS}^\cap \) against an implementation behaving as an \( \text{IOTS}^\cap \) are independent of the communication scheme. We are now in a position to establish the correctness of Theorem 6.3. We first recall the theorem and afterwards, we provide its proof:

**Theorem 6.3.** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an arbitrary \( \text{IOTS}^\cap \) and let \( \langle T, U \cup \{ \theta \}, I, \rightarrow, \bar{t} \rangle \) be a \( \text{TTS}^\cap \). If \( \bar{s} \) is non-divergent, then \( \bar{s} \) passes \( \bar{t} \) iff \( Q(\bar{s}) \) passes \( \bar{t} \).

**Proof.** We prove the bi-implication that \( \bar{s} \) passes \( \bar{t} \) iff \( Q(\bar{s}) \) passes \( \bar{t} \) by contraposition. Let language \( L' = I \cup U \cup \{ \theta \} \).

- To prove the left-to-right implication, we suppose, that not \( Q(\bar{s}) \) passes \( \bar{t} \). By Definition 2.34 and Proposition 6.9, \( \bar{t} \}Q(\bar{s}) \Rightarrow \text{fail} \}Q(s') \), for some \( \sigma' \in L'^\cap \) and \( s' \in S \). As a result of Lemma 6.11, there is a non-empty set \( S \subseteq \{s'' \in S \mid s' \xrightarrow{\epsilon} s''\} \) such that for all \( s'' \in S \), \( \bar{t} \}s \xrightarrow{\sigma'} \text{fail} \}s'' \), which was what we needed to prove.

- Similar to the above case, the right-to-left implication is proved by contraposition. We assume, that not \( \bar{s} \) passes \( \bar{t} \). Then there are \( \sigma' \in L'^\cap \) and \( s'' \in S \), \( \bar{t} \}s \xrightarrow{\sigma'} \text{fail} \}s'' \). Using Property 2.32 leads to \( \bar{t} \}Q(\bar{s}) \Rightarrow \text{fail} \}Q(s'') \).

\( \square \)

By Property 4.30, we know that internal-choice test cases check the internal-choice traces of a given specification \( \bar{s} \). We showed in Section 4.4 that test cases that are generated by Algorithm 4.4.1 from the specification \( \bar{s} \) and \( \text{IC traces}(\bar{s}) \) are internal-choice test cases. Furthermore, they are sound and exhaustive with respect to the implementation
relation $\text{ioco}_{\text{ICtraces}(i)}$, see Corollary 4.41; they assess if an implementation behaving as an IOTSM relates to an IOLTS specification with respect to the implementation relation $\text{ioco}_{\text{ICtraces}(i)}$. Combining Theorem 6.3 with Corollary 4.41, we find the following corollary.

**Corollary 6.12.** Let specification $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an arbitrary (non-divergent) IOLTS, and let $\langle R, I, U, \rightarrow, \bar{r} \rangle$ be an arbitrary (non-divergent) IOTSM. Then we have,

$$\bar{r} \text{ioco}_{\text{ICtraces}(i)} \bar{s} \iff Q(\bar{r}) \text{ioco}_{\text{ICtraces}(i)} \bar{s}$$

Presuming that the class of specifications is restricted to IOTSM’s, we find as a direct consequence of the above corollary that $\bar{r} \text{ioco}_{\text{ICtraces}(i)} \bar{s}$ if $Q(\bar{r}) \text{ioco}_{\text{ICtraces}(i)} \bar{s}$. We showed in Theorem 4.28 that the $\text{ioco}$ relation coincides with the $\text{ioco}_{\text{ICtraces}(i)}$ relation in the synchronous setting. Now, one can ask if the result of Theorem 4.28 is still valid in the asynchronous setting. The answer to this question with respect to the above corollary clarifies if adapting the classes of implementations, specifications, and test cases to IOTSM’s provides a solution to use the $\text{ioco}$ relation in the queue context when test cases are derived from the specification, not its composition with queues. Interestingly, we show in the next section that the answer to this question will be negative that motivates our study in the rest of this chapter.

### 6.2 Restricting Implementation Models

In this section, we re-cast the results of the previous section to the setting with standard $\text{ioco}$ test cases, i.e., non-internal choice test cases. We first show that the result of Theorem 4.28 cannot be trivially generalized to the asynchronous setting. Thus we cannot reuse our result in the previous section in conformance testing based on $\text{ioco}$ relation. Using an approach inspired by [Tre92, Chapter 5] and [SP10], we show how to re-formulate Theorem 6.3 in this setting.

In section 4.1 it is shown that restricting the set of traces $F$ in implementation relation $\text{ioco}_{F}^{a,b}$ will lead to a weaker testing power. Yet, we proved in Theorem 4.28 that the discrimination power of $\text{ioco}_{\text{Straces}(i)}^{a,b}$ for a given specification $\bar{s}$ does not decrease by examining internal-choice traces of $\bar{s}$ instead of suspension traces in setting $a, b \in \{\|\}$, i.e., for any IOTSM $\bar{i}$, we have $\bar{i} \text{ioco}_{\text{Straces}(i)}^{\|} \bar{s}$ if $\bar{i} \text{ioco}_{\text{ICtraces}(i)}^{\|} \bar{s}$. However, in the following example, we demonstrate that the testing power of $\text{ioco}_{\text{ICtraces}(i)}^{\|}$ and $\text{ioco}_{\text{Straces}(i)}^{\|}$ are different in the asynchronous setting, i.e., for an IOTSM $\bar{i}$, we have that $Q(\bar{i}) \text{ioco}_{\text{Straces}(i)}^{\|} \bar{s}$ while $Q(\bar{i}) \text{ioco}_{\text{ICtraces}(i)}^{\|} \bar{s}$.

**Example 6.13.** IOLTS $\bar{i}$ in Figure 6.4 on page 115 shows a test case derived from IOTS $\bar{s}$ in Figure 6.4, which is an internal-choice IOTS. Assume that at the same time $\bar{s}$ is also used as the implementation. For $\bar{s}$ as specification and implementation, we have that $\bar{s} \text{ioco} \bar{s}$. However, we can reach a fail verdict for $\bar{s}$ under the queue context when using the test case $\bar{i}$. Consider the sequence ‘coin button refund’; in the queue context, the execution $\bar{i} \| Q(\bar{s}) \xrightarrow{\text{coin}} t_{1} |_{\bar{s} \in \text{coin}} \xrightarrow{\bar{s}} t_{1} | Q(s_{1}) \xrightarrow{\text{button}} t_{2} | _{\text{refund}} \xrightarrow{\text{refund}} \text{fail} |_{\bar{s} \in \text{button}}$ is possible, which leads to the fail state; thus, $Q(\bar{s}) \text{ioco} \bar{s}$.  

114
6.2. Restricting Implementation Models

On the other hand, we obtain from $\bar{s}_{\text{ioco}}$ and Theorem 4.28 that $\bar{s}_{\text{ioco}}\cap_{\text{ICtraces}(s)} \bar{s}$. We consequently find from Theorem 6.3 that $Q(\bar{s})_{\text{ioco}}\cap_{\text{ICtraces}(s)} \bar{s}$. This shows that Theorem 6.3 cannot be trivially generalized to the ioco setting (even when excluding divergence and allowing for non-input-enabled states).

Our main interest in this section is to investigate implementations for which ioco test cases cannot distinguish between synchronous and asynchronous modes of testing. To this end, we consider the relation between traces of a system and those of the system in the queue context. As we observed so far, the traces of a system in the synchronous setting are different from those in the asynchronous setting. Generally, communicating with the IUT through queues can introduce a reordering of actions in the traces of the IUT to an external observer (a tester). While the IUT produces outputs and fills the output queue, the tester can provide inputs and put them in the input queues. So those input actions of the input queue precede the output actions which are in the output queue, from the viewpoint of the tester. However, the IUT executes those inputs and outputs in the reverse order. To formally reason about the observable behavior of a system in the queue context, we formalize the reordering of inputs and outputs occurring in the queue context. To this end, we first define the delay operator @, given below, that formally characterizes traces obtained by preceding inputs to outputs.

**Definition 6.14** (Delay relation). Let $L$ be a finite alphabet partitioned in $I$ and $U$. The delay relation $@ \subseteq L^* \times L^*$ is defined by the following deduction rules:

\[
\frac{\rho_i, \sigma_i \in I^* \quad \sigma_u \in U^*}{\rho_i \sigma_u \sigma_i \@ \rho_i \sigma_i \sigma_u} \quad \text{PUSH} \quad \frac{\sigma \@ \sigma' \quad \rho \@ \rho'}{\sigma \rho \@ \sigma' \rho'} \quad \text{COM}
\]

**Example 6.15.** Consider IOTS $\bar{s}$ in Figure 6.4. Observing the transitions $Q(\bar{s})$ in the queue context, we find that sequence $\sigma \in \text{Straces}(Q(\bar{s}))$ with $\sigma = \text{coin button refund}$. However, the actual sequence executed by

---

Figure 6.4: An internal-choice model of a vending machine and an ioco test case of that machine
IOTS $\bar{s}$ is sequence $\sigma' = \text{coin refund button}$. Using deduction rule $\text{PUSH}$ in Definition 6.14, we find that $\sigma'@\sigma$. In fact, deduction rule $\text{PUSH}$ formalizes the situation in which a sequence of inputs is provided in advance and is put in the input queues, as illustrated in the above transitions.

Using the delay operator, we can precisely characterize the externally observable behavior of a system in the queue context with respect to its actual behavior. In fact, every trace of a system behaving as an IOTS in the queue context is retrievable from some actual trace of the system where inputs may be shifted before outputs. Before we characterize the formal relation between traces of a system in the queue context with its traces in the synchronous setting, we first prove the following lemmas which are essential in establishing a link between traces in the synchronous and asynchronous settings. The following lemma states that all inputs provided by a tester are accepted by a system behaving as an IOTS, even in the asynchronous setting.

**Lemma 6.16.** Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOTS over language $L = I \cup U$, and let $s \in S$ and $\sigma \in L^*_S$. Then $\sigma \in \text{Straces}(Q(s))$ implies that there is a $s' \in S$ such that $Q(s) \Rightarrow Q(s')$.

**Sketch of the proof.** According to the delay operator, observing quiescence in the queue context cannot be delayed. Therefore, the trace $\sigma$ can be chopped into smaller sub-traces with respect to occurrences of $\delta$. The correctness of the above thesis for each of these sub-traces can easily be shown by applying the inference rules in Definition 2.11. Finally, the proof of the above thesis for the whole trace $\sigma$ is given by using an induction on the number of $\delta$’s in $\sigma$.

We are now in a position to formally establish the link between traces in the asynchronous setting with those in the synchronous setting. 6.18 as given below.

**Proposition 6.18.** Let $\langle S, I, U, \rightarrow, \bar{s} \rangle$ be an IOTS over language $L = I \cup U$. Let $s \in S$ and $\sigma \in L^*_S$. Then $\sigma \in \text{Straces}(Q(s))$ implies there is a $s' \in \text{Straces}(s)$ such that $s \Rightarrow s'$ and $\sigma'@\sigma$.

**Proof.** Using the lemmata given above, the proof of the statement follows from the observations below. We have that $\sigma \in \text{Straces}(Q(s))$, implying that $\exists s' \in S \cdot Q(s) \Rightarrow Q(s')$, due to Lemma 6.16. It follows from the previous observation and Lemma 6.17 that $\exists \sigma' \in \text{Straces}(s) \cdot s \Rightarrow s'$ and $\sigma'@\sigma$ which was to be shown.
To be sure that the test context has no effect on the verdicts delivered by \textit{ioco-test} cases, it suffices to prevent the occurrences of false \textit{fail} verdicts. More precisely, whenever a test execution reaches the \textit{fail} verdict in the asynchronous setting, it has to be guaranteed that the test execution will also reach the \textit{fail} verdict in the synchronous setting. That can intuitively be assured if any test run in the queue context can be simulated in the synchronous setting. In this sense, we define a subclass of IOTSs, given below, in which traces observed in the queue context are indistinguishable from those observable in the synchronous setting.

\textbf{Definition 6.19} (Delay right-closed IOTS). Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOTS over language \( L = I \cup U \). A set \( L' \subseteq L_\delta^* \) is delay right-closed iff for all \( \sigma \in L' \) and \( \sigma' \in L_\delta^* \), if \( \sigma \circ \sigma' \) then \( \sigma' \in L' \). The IOTS \( \bar{s} \) is delay right-closed iff \( \text{Straces}(\bar{s}) \) is delay right-closed.

We denote the class of delay right-closed IOTSs ranging over \( I \) and \( U \) by \( \text{IOTS}^@ (I, U) \). The property below gives a characterization of the traces of delay right-closed IOTSs.

\textbf{Property 6.20}. Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOTS. The IOTS \( \bar{s} \) is delay right-closed if for all \( \sigma \in L_\delta^* \), all \( x \in U \) and \( a \in I \), we have:

\[ \sigma xa \in \text{Straces}(\bar{s}) \text{ then } \sigma ax \in \text{Straces}(\bar{s}) \]

\textbf{Example 6.21}. Consider the IOTS \( \bar{s} \) given in Figure 6.5. It is not hard to check that \( \bar{s} \) is delay right-closed.

\textbf{Proposition 6.22}. Let IOTS \( \bar{s} \) be delay right-closed. Then, \( \text{Straces}(Q(\bar{s})) = \text{Straces}(\bar{s}) \).

\textit{Proof}. We divide the proof obligation into two parts: \( \text{Straces}(Q(\bar{s})) \subseteq \text{Straces}(\bar{s}) \) and \( \text{Straces}(\bar{s}) \subseteq \text{Straces}(Q(\bar{s})) \). It is not hard to verify that the latter holds vacuously, even for arbitrary IOTSs.

It therefore remains to show that \( \text{Straces}(Q(\bar{s})) \subseteq \text{Straces}(\bar{s}) \). Consider a trace \( \sigma \in \text{Straces}(Q(\bar{s})) \); by Proposition 6.18, \( \exists \sigma' \in \text{Straces}(\bar{s}) \bullet \sigma' @ \sigma \). As \( \bar{s} \) is delay right-closed, we obtain the required \( \sigma \in \text{Straces}(\bar{s}) \). \( \Box \)

As stated in the following theorem, the verdicts obtained by executing an arbitrary test case on a delay right-closed IOTS do not depend on the execution context. That is, the verdict does not change when the communication between the implementation and the test case is synchronous or asynchronous.
The correctness of the above thesis follows from the existence of a sound and complete test suite with respect to the ioco context.

Theorem 6.24. Let \( \langle R, I, U, \rightarrow_r, \vec{r} \rangle \) be a delay right-closed IOTS and let IOLTS \( \langle S, I, U, \rightarrow_s, \vec{s} \rangle \) be a specification. Then \( \vec{r} \) ioco \( \vec{s} \) iff \( Q(\vec{r}) \) ioco \( Q(\vec{s}) \).

Proof. The correctness of the above thesis follows from the existence of a sound and complete test suite with respect to the ioco relation along with Theorem 6.23.

We now show that being delayed right-closed for implementations is also a necessary condition to ensure the same verdict in the synchronous and the asynchronous setting.

Theorem 6.25. Let \( \langle R, I, U, \rightarrow_r, \vec{r} \rangle \) be an IOTS. If for every test case \( \langle T, U \cup \{ \theta \}, I, \rightarrow_t, \vec{t} \rangle \), we have \( \vec{r} \) passes \( \vec{t} \) \( \Leftrightarrow \) \( Q(\vec{r}) \) passes \( \vec{t} \), then \( \vec{r} \) is a delay right-closed IOTS.

Proof. We prove the theorem by contraposition, i.e., we show that if we test a non-delay right-closed IOTS, there is a test case that gives a pass verdict in the synchronous setting but a fail verdict in the asynchronous setting.

Let \( \langle R, I, U, \rightarrow_r, \vec{r} \rangle \) be an IOTS that is not delay right-closed. Thus, there is some \( x \in U \), \( a \in I \) such that \( \sigma xa \in \text{Straces}(\vec{r}) \), but not \( \sigma ax \in \text{Straces}(\vec{r}) \). Let \( \langle T, U \cup \{ \theta \}, I, \rightarrow_t, \vec{t} \rangle \) be a test case such that there is a \( t' \in T \) satisfying:

1. \( \vec{t} \xrightarrow{\sigma} t' \),

2. \( t' \xrightarrow{a} t'' \), and \( t'' \xrightarrow{\sigma x} \text{fail} \).

3. for all \( \sigma' \) such that \( \vec{t} \xrightarrow{\sigma'} \text{fail} \) we have \( \sigma' = \sigma ax \).

Observe that the existence of such a test case is immediate. Then there are \( \sigma_i \in I^* \), \( \sigma_u \in U^* \) and a state \( r \in (\vec{r} \text{ after } \sigma) \) such that \( \vec{t} \xrightarrow{\sigma} \text{fail} \xrightarrow{\sigma ax} \text{fail} \xrightarrow{\sigma ax} \text{fail} \) \( [\sigma_u \ll r \ll \sigma, a] \), i.e., not \( Q(\vec{r}) \) passes \( \vec{t} \). However, we do not have \( \vec{t} \xrightarrow{\sigma} \text{fail} \). By the construction of the test case, we find that \( \vec{r} \) passes \( \vec{t} \).
6.3 Restricting Specification Models

In the previous section, we identified a class of implementations whose context of test execution is immaterial to the tester: the result of testing is the same in both synchronous and asynchronous settings with respect to the \texttt{ioco} relation. Similar to the approach introduced in [WW09; Wei09], those conditions imposed on implementations are semantic. Since in \texttt{ioco} conformance testing theory, the IUT is treated as a black box, those conditions thus cannot be directly verified on the IUT at testing time.

In this section, we investigate how we can re-cast our results in the previous section to the class of specifications which are available at testing time. Our main interest in this section thus is to identify a subclass of specifications for which the \texttt{ioco}-test cases always reach sound verdicts even when the IUT is tested using asynchronous interactions.

Outputs of a reactive system are defined as actions that are fully under the control of the system itself, unlike inputs which are controlled by the environment (the tester), see Section 2.1.2. Hence, in states where both an input and an output are enabled (henceforth referred to as race situation), there can be a competition between the IUT and the tester to send out an output or to provide an input, respectively.

**Definition 6.26 (Race situation).** Let \( \langle S, I, U, \rightarrow, \vec{s} \rangle \) be an IOLTS, and let the sequence \( \sigma \) be a suspension trace of \( \vec{s} \), i.e., \( \sigma \in \text{Straces}(\vec{s}) \). We say that \( \vec{s} \) reaches a race situation after executing \( \sigma \), if and only if,

\[
\exists a \in I, x \in U \cdot \sigma a \in \text{Straces}(\vec{s}) \land \sigma x \in \text{Straces}(\vec{s})
\]

In a race situation in the synchronous setting, the IUT and the tester are able to synchronize on the same action; they both execute either an input or an output. However, in the asynchronous setting that synchronization is impossible, due to unknown lags in communications. Therefore, in race situations in the asynchronous setting, the IUT may produce an output while the tester is providing an input. Therefore, from the viewpoint of the tester the order of execution of inputs and outputs is different from the actual order executed by the IUT in a same test execution.

**Example 6.27.** Consider IOTS \( \vec{r} \) depicted in Figure 6.3. After producing output \( p_{rq} \), \( \vec{r} \) reaches state \( r' \) that is a race situation. As explained in Example 6.4, since the moment of choice between input \( p_{rs} \) and output \( r_{rq} \) at state \( r' \) is not observable by an external observer in the queue context, the tester may provide input \( p_{rs} \) while IOTS \( \vec{r} \) produces output \( r_{rq} \) which eventually leads to the \texttt{fail} verdict.

Concerning race situations, in the remainder of this section, we identify a subset of IOLTSs whose race situations have no effect on testing when interactions with the IUT are carried out asynchronously. More precisely, the order of executing inputs and outputs at race situations in testing with those specifications makes no difference in the test result; it does not matter whether an input is first executed or an output. We next show that test verdicts delivered by \texttt{ioco}-test cases generated from such a specification are independent of the test execution context. The example given below explains the intuition behind that subclass of specifications that we are going to characterize in this section.

**Example 6.28.** Consider IOLTSs depicted in Figure 6.6. After executing output \( p_{rq} \), both IOLTSs \( \vec{m} \) and \( \vec{s} \) end up at race situations where they can choose between input \( p_{rs} \)
and output \( r_{rq} \). Output \( r_{rq} \) is not acceptable in IOLTS \( \hat{m} \) after input transition \( p_{rs} \) is executed at the corresponding race situation; thus, the choice made in IOLTS \( \hat{m} \) at the race situation affects future observations. However, that does not hold for IOLTS \( \hat{s} \), since both sequences \( p_{rs} r_{rq} \) and \( r_{rq} p_{rs} \) are acceptable; thus, the order of execution of input \( p_{rs} \) and output \( r_{rq} \) does not matter.

Before formally defining the desired subclass of specifications, we first introduce an \textit{ioco}-like relation between two sets of states of an IOLTS which is essential for characterizing those specifications.

**Definition 6.29** (Input-output simulation). Let \( \langle S, I, U, \rightarrow, \hat{s} \rangle \) be an IOLTS over language \( L = I \cup U \). Let \( S' \subseteq S \) and \( P' \subseteq P \) be two subsets of states. We say \( P' \) is in input-output simulation relation with \( S' \), denoted by \( P' \text{ios} S' \), if and only if for any \( \sigma \in \text{Straces}(S') \), the two following conditions hold:

1. \( \sigma \in \text{Straces}(P') \Rightarrow \text{out}(P' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \)
2. \( \forall \sigma', \sigma'' \in L_{\hat{s}}, a \in I \bullet \sigma = \sigma' \sigma'' \wedge \sigma' \in \text{Straces}(P') \wedge \sigma'' \notin \text{Straces}(P') \Rightarrow U \cup \{ \delta \} \subseteq \text{out}(S' \text{ after } \sigma) \)

**Remark 6.30.** Let \( \langle S, I, U, \rightarrow, \hat{s} \rangle \) and \( \langle R, I, U, \rightarrow, \hat{r} \rangle \) be two IOLTSs. We define the universal IOLTS \( \langle M, I, U, \rightarrow, \hat{m} \rangle \) as follows,

- \( M = S \cup R \cup \{ \hat{m} \} \)
- \( \rightarrow = \rightarrow_s \cup \rightarrow_r \cup \{ (\hat{m}, \tau, \hat{s}), (\hat{m}, \tau, \hat{r}) \} \)

Informally, Definition 6.29 states that two subset of states \( P' \) and \( S' \) are in the \textit{ios} relation, i.e., \( P' \text{ios} S' \) if and only if after executing any suspension traces of \( S' \), outputs of \( P' \) are among those specified in \( S' \). The input-restriction of \( P' \) gets weaker in comparison with the \textit{ioco} relation and \( P' \) is not necessarily assumed to be input enabled; \( P' \) cannot always simulate all input actions of \( S' \). When an input action of \( S' \) is not simulated by \( P' \), all outputs (including quiescence) are allowed after that input in \( S' \). Consequently, the behavior of system after an input which is only specified in \( S' \) but in \( P' \) does not have any impact on the test results.
Example 6.31. Consider two IOLTSs depicted in Figure 6.6 with inputs \( I = \{p_{rs}\} \) and outputs \( U = \{p_{rq}, r_{rq}\} \). We construct the universal IOLTS \( \hat{\hat{u}} \) from IOLTS’s \( \hat{m} \) and \( \hat{s} \), as mentioned above. Take two sets \( \{\hat{m}\} \) and \( \{\hat{s}\} \) as two subsets of the whole state of universal IOLTS \( \hat{\hat{u}} \). It is not difficult to verify that \( \{\hat{m}\} \text{ ios} \{\hat{s}\} \). For instance, consider the sequence \( \sigma' = p_{rq} r_{rq} \). Although input \( p_{rs} \) is not specified in IOLTS \( \hat{m} \) after sequence \( \sigma \), i.e., \( p_{rs} \notin \text{Sinit}(\hat{m}''') \), \( U \cup \{\delta\} \notin \text{out}(\hat{s}'') \) for \( s'' = \hat{s} \) after \( \sigma p_{rs} \). Thus, the second condition in Definition 6.29 is violated. We therefore find that \( \{\hat{m}\} \text{ ios} \{\hat{s}\} \). Now consider \( \sigma' = p_{rq} p_{rs} \). We observe that \( \{m'\} = \hat{m} \) after \( \sigma' \) and \( \{s',s\} = \hat{s} \) after \( \sigma' \). Clearly, \( \text{out}(\{m'\}) \text{ after } \sigma' \subseteq \text{out}(\{s',s\} \text{ after } \sigma') \) for all \( \sigma' \in \text{Straces}(\{s',s\}) \). Thus, we find that \( \{m'\} \text{ ios} \{s',s\} \).

As a direct consequence of Definition 6.29, we find the following property. The property below extends the second condition of the above definition over a sequence of input actions.

Property 6.32. Let \( \langle S, I, U, \rightarrow, s \rangle \) be an IOLTS over language \( L = I \cup U \), and let \( P' \subseteq P \) and \( S' \subseteq S \) be two arbitrary sets of states such that \( P' \text{ ios} S' \). Let \( \sigma \in \text{Straces}(S') \cap \text{Straces}(P') \), and let \( a \in I \) such that \( a \in \text{Sinit}(S' \text{ after } \sigma) \). Then, \( a \notin \text{Sinit}(P' \text{ after } \sigma) \) implies that

\[
\forall \sigma' \in L_S^* \bullet \sigma' \in \text{Straces}(S' \text{ after } \sigma a) \Rightarrow U \cup \{\delta\} \subseteq \text{out}(S' \text{ after } \sigma a \sigma')
\]

We show in the proposition below that the \( \text{ios} \) relation is a preorder, which is essential for our results in the rest of this section.

Proposition 6.33. Let \( \langle S, I, U, \rightarrow, s \rangle \) be an IOLTS over language \( L = I \cup U \). The relation \( \text{ios} \subseteq P(S) \times P(S) \) is a preorder relation.

Proof. To prove that the \( \text{ios} \) relation is a preorder, we need to show that it is reflexive and transitive. We take an arbitrary set of states \( S' \subseteq S \). Clearly, \( \text{out}(S' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \) for any \( \sigma \in \text{Straces}(S') \). From Definition 6.29, we obtain that \( S' \text{ ios} S' \). Therefore \( \text{ios} \) relation is reflexive.

We assume three arbitrary sets of states \( P' \subseteq S, Q' \subseteq S \), and \( S' \subseteq S \) such that \( P' \text{ ios} S' \) and \( Q' \text{ ios} P' \). To prove the transitivity property of the relation \( \text{ios} \), we have to show that \( Q' \text{ ios} S' \). Following Definition 6.29, we need to show that the two conditions below are fulfilled for any \( \sigma \in \text{Straces}(S') \),

- \( \sigma \in \text{Straces}(Q') \Rightarrow \text{out}(Q' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \)
- \( \forall \sigma', \sigma'' \in L_S^*, a \in I \bullet \sigma = \sigma'a\sigma'' \land \sigma' \in \text{Straces}(Q') \land \sigma'a \notin \text{Straces}(Q') \Rightarrow U \cup \{\delta\} \subseteq \text{out}(S' \text{ after } \sigma) \)

We take an arbitrary sequence \( \sigma \in \text{Straces}(S') \). We distinguish four different cases as follows:

- We assume that \( \sigma \in \text{Straces}(P') \) and \( \sigma \in \text{Straces}(Q') \). Following \( P' \text{ ios} S' \) and \( Q' \text{ ios} P' \), we get that \( \text{out}(P' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \) and \( \text{out}(Q' \text{ after } \sigma) \subseteq \text{out}(P' \text{ after } \sigma) \). We obtain from the transitivity property of the subset relation on sets that \( \text{out}(Q' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \).
• We assume that \( \sigma \in \text{Straces}(P') \) and \( \sigma \notin \text{Straces}(Q') \). Therefore, \( \sigma = \sigma' a \sigma'' \) such that \( \sigma' \in \text{Straces}(Q') \) while \( \sigma'a \notin \text{Straces}(Q') \) with \( \sigma', \sigma'' \in L_5^\delta \) and \( a \in L_5 \).

With respect to Definition 6.34, we just need to investigate the case \( a \in I \); thus \( \sigma' \in \text{Straces}(Q') \) and \( \sigma'a \notin \text{Straces}(Q') \) with \( a \in I \). Following \( Q' \text{ios} P' \) along with \( a \in I \), we obtain that \( U \cup \{\delta\} \subseteq \text{out}(P' \text{ after } \sigma') \). On the other hand, we know from \( P' \text{ios} S' \) and \( \sigma \in \text{Straces}(P') \) that \( \text{out}(P' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \). Combining the last two observations, we conclude that \( U \cup \{\delta\} \subseteq \text{out}(S' \text{ after } \sigma) \) which was to be shown.

• We assume that \( \sigma \notin \text{Straces}(P') \) and \( \sigma \in \text{Straces}(Q') \). Thus, \( \sigma = \sigma' a \sigma'' \) with \( \sigma', \sigma'' \in L_5^\delta \) and \( a \in L_5 \) such that \( \sigma' \in \text{Straces}(P') \) and \( \sigma'a \notin \text{Straces}(P') \). We know from Definition 6.29 and \( Q' \text{ios} P' \) that \( \text{out}(Q' \text{ after } \sigma') \subseteq \text{out}(P' \text{ after } \sigma') \).

Since \( \sigma a \in \text{Straces}(Q') \), we find from the previous observation that \( a \in I \), because \( \text{out}(Q' \text{ after } \sigma') \notin \text{out}(P' \text{ after } \sigma') \). Following Definition 6.29 and \( a \in I \), we conclude from \( P' \text{ios} S' \) that \( U \cup \{\delta\} \subseteq \text{out}(S' \text{ after } \sigma) \). Therefore, the last observation along with \( \text{out}(Q' \text{ after } \sigma) \subseteq U \cup \{\delta\} \) results in \( \text{out}(Q' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \).

• We assume that \( \sigma \notin \text{Straces}(P') \) and \( \sigma \notin \text{Straces}(Q') \). Thus, \( \sigma = \sigma' a \sigma'' \) with \( \sigma', \sigma'' \in L_5^\delta \) and \( a \in L_5 \) such that \( \sigma' \in \text{Straces}(P') \) and \( \sigma'a \notin \text{Straces}(P') \). We proceed the proof by distinguishing between \( a \in U \cup \{\delta\} \) and \( a \in I \).

  - We assume that \( a \in U \cup \{\delta\} \); thus \( a \notin \text{out}(P' \text{ after } \sigma') \). We now distinguish two cases.

    * We assume that \( \sigma' \in \text{Straces}(Q') \). Following \( Q' \text{ios} P' \), we obtain that \( \text{out}(Q' \text{ after } \sigma) \subseteq \text{out}(P' \text{ after } \sigma') \). Therefore, \( a \notin \text{out}(Q' \text{ after } \sigma) \) is obtained which implies that \( \sigma'a \notin \text{Straces}(Q') \). With respect to Definition 6.29, we do not need to investigate this case.

    * We assume that \( \sigma' \notin \text{Straces}(Q') \). Therefore, \( \sigma' = \rho x P' \) with \( \rho, P' \in L_5^\delta \) and \( x \in L_\delta \) such that \( \rho \in \text{Straces}(Q') \) and \( \rho x \notin \text{Straces}(Q') \). With respect to Definition 6.29, we just need to consider the case where \( x \in I \). Thus, \( \rho \in \text{Straces}(Q') \) and \( \rho x \notin \text{Straces}(Q') \) for \( x \in I \). Following \( Q' \text{ios} P' \), we obtain that \( U \cup \{\delta\} \subseteq \text{out}(P' \text{ after } \sigma') \) which contradicts \( a \notin \text{out}(P' \text{ after } \sigma) \).

  - We suppose that \( a \in I \). We obtain from \( P' \text{ios} Q' \) that \( U \cup \{\delta\} \subseteq \text{out}(S' \text{ after } \sigma) \). We find from \( \text{out}(Q' \text{ after } \sigma) \subseteq U \cup \{\delta\} \) that \( \text{out}(Q' \text{ after } \sigma) \subseteq \text{out}(S' \text{ after } \sigma) \), which was to be shown.

\[ \square \]

We are now in the position to formally define a subclass of IOLTSs whose race situations are externally indistinguishable in the asynchronous setting. Therefore, \textit{ioco}-test cases generated from such specifications always deliver verdicts which are meaningful for the IUT even when interactions between the tester and the IUT proceed asynchronously.

**Definition 6.34 (Robust IOLTS).** Let IOLTS \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be a specification. We say specification \( \bar{s} \) is \textit{robust} iff for any sequence \( \sigma \in \text{Straces}(\bar{s}) \) and for any input \( a \in I \) and output \( x \in U \) for which \( \{a, x\} \subseteq \text{Sinit}(\bar{s} \text{ after } \sigma) \), also:

\[ 122 \]
1. \( x \in \text{out}(s \text{ after } \sigma a) \)

2. \( a \in \text{Sinit}(s \text{ after } \sigma x) \Rightarrow (s \text{ after } \sigma xa) \text{ios}(s \text{ after } \sigma ax) \)

3. \( a \notin \text{Sinit}(s \text{ after } \sigma x) \Rightarrow \forall \sigma' \in \text{Straces}(s \text{ after } \sigma ax) \bullet \text{out}(s \text{ after } \sigma ax\sigma') = U \cup \{\delta\} \)

**Example 6.35.** Consider IOLTS \( \bar{m} \) depicted in Figure 6.6 on page 120. As explained before, it models a part of the purchase scenario in which the switch sends a purchase request to the core banking system and either receives a response, or after an internal time-out (modeled by an internal transition) sends a reversal request. We find that IOLTS \( \bar{m} \) is not (asynchronously) robust. Observe that sequence \( p_{rq} \) leads to a race situation in specification \( \bar{m} \) where both input and output transitions \( p_{rs} \) and \( r_{rq} \) are enabled. However, output \( r_{rq} \) is not acceptable in specification \( \bar{m} \) after the sequence \( p_{rq} p_{rs} \).

Consider now IOLTS \( \bar{s} \) depicted in Figure 6.6. It is similar to the IOLTS \( \bar{m} \), except that after the internal time-out, it accepts the late purchase response. The late purchase response may be received either before or after sending the reversal request. By checking the subset of states \( s \text{ after } p_{rq} \) as the only race situation in IOLTS \( \bar{s} \), we find that IOLTS \( \bar{s} \) is asynchronously robust. Observe that an external observer cannot distinguish between executing sequences \( p_{rq} p_{rs} r_{rq} \) and \( p_{rq} p_{rs} r_{rq} \) in IOLTS \( \bar{s} \).

The theorem below states that the testing power of the \text{ioco} relation for robust IOLTS in the asynchronous setting is the same as one in the synchronous setting.

**Theorem 6.36.** Let IOLTS \( \bar{s} \) be a robust specification, and IOTS \( \bar{r} \) be an implementation. Then

\( \bar{r} \text{ioco} \bar{s} \iff Q(\bar{r}) \text{ioco} \bar{s} \)

Before we address the proof of the above theorem, we first show the correctness of the proposition below, which is a direct consequence of the definition of robust specifications.

**Proposition 6.37.** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be a robust IOLTS, and let \( \sigma \in \text{Straces}(\bar{s}) \) such that \( \bar{s} \) reaches a race situation after executing \( \sigma \). Let sequence \( \rho \in I^+ \), and let \( x \in U \). Then, the following conditions hold if \( \{x, \rho\} \subseteq \text{Straces}(\bar{s} \text{ after } \sigma) \),

1. \( x \in \text{out}(s \text{ after } \sigma \rho) \)

2. \( \rho \in \text{Straces}(s \text{ after } \sigma x) \Rightarrow (s \text{ after } \sigma x\rho) \text{ios}(s \text{ after } \sigma x\rho x) \)

3. \( \rho \notin \text{Straces}(s \text{ after } \sigma x) \Rightarrow \forall \sigma' \in \text{Straces}(s \text{ after } \sigma x\rho) \bullet \text{out}(s \text{ after } \sigma x\rho x\sigma') = U \cup \{\delta\} \)

**Proof.** We suppose that \( \{x, \rho\} \subseteq \text{Straces}(s \text{ after } \sigma) \); thus, we need to show that the required conditions hold. The proof proceeds by induction on the length of \( \rho \).

- **Base case.** We assume for the induction basis that the length of \( \rho \) is one, i.e., \( \rho = a \) with \( a \in I \). Clearly, the thesis holds with respect to Definition 6.34.

- **Induction step.** We assume that the given thesis holds for all input sequences of length less than \( n \) and that the length of \( \rho \) is \( n > 1 \); thus \( \rho = \rho' a \) where the length of \( \rho' \in I^+ \) is \( n - 1 \) and \( a \in I \). We first show that the first condition is
satisfied, i.e., \( x \in \text{out}(\bar{s} \text{ after } \sigma \rho' a) \). From the induction hypothesis, we obtain that \( x \in \text{out}(\bar{s} \text{ after } \sigma \rho') \). With respect to Definition 6.34, we conclude from \( \{x, a\} \in \text{Straces}(\bar{s} \text{ after } \sigma \rho') \) with \( x \in U \) and \( a \in I \) that \( x \in \text{out}(\bar{s} \text{ after } \sigma \rho' a) \). Thus \( x \in \text{out}(\bar{s} \text{ after } \sigma \rho) \) which was to be shown.

We now distinguish two cases; \( \rho \in \text{Straces}(\bar{s} \text{ after } \sigma x) \) and \( \rho \notin \text{Straces}(\bar{s} \text{ after } \sigma x) \). We investigate each of these conditions separately.

- We assume that \( \rho \in \text{Straces}(\bar{s} \text{ after } \sigma x) \). Thus, we have to show that the second condition is fulfilled, i.e., \( (\bar{s} \text{ after } \sigma \rho \rho) \text{ios}(\bar{s} \text{ after } \sigma \rho x) \). From the induction hypothesis, we know that \( x \in \text{out}(\bar{s} \text{ after } \sigma \rho') \). We distinguish between two cases: \( a \in \text{Sinit}(\bar{s} \text{ after } \sigma \rho' x) \) and \( a \notin \text{Sinit}(\bar{s} \text{ after } \sigma \rho' x) \).

  * We assume that \( a \in \text{Sinit}(\bar{s} \text{ after } \sigma \rho' x) \). For \( x \in U \) and \( a \in I \), we know that \( \{x, a\} \in \text{Straces}(\bar{s} \text{ after } \sigma \rho') \). From Definition 6.34, we deduce that \( (\bar{s} \text{ after } \sigma \rho' x a) \text{ios}(\bar{s} \text{ after } \sigma \rho x) \). Following the induction hypothesis, we also obtain that \( (\bar{s} \text{ after } \sigma \rho' x a) \text{ios}(\bar{s} \text{ after } \sigma \rho' x) \). Consequently, \( (\bar{s} \text{ after } \sigma \rho' x a) \text{ios}(\bar{s} \text{ after } \sigma \rho' x a) \) is obtained from Definition 6.34. Following Proposition 6.3, we deduce from the two above results that \( (\bar{s} \text{ after } \sigma \rho' x a) \text{ios}(\bar{s} \text{ after } \sigma \rho' x a) \), which was to be shown.

  * We assume that \( a \notin \text{Sinit}(\bar{s} \text{ after } \sigma \rho' x) \), whereas for \( x \in U \) and \( a \in I \) we know that \( \{x, a\} \in \text{Straces}(\bar{s} \text{ after } \sigma \rho') \). From Definition 6.34, we obtain that \( U \cup \{\delta\} \subseteq (\bar{s} \text{ after } \sigma \rho' x a x) \) for any \( \sigma' \in \text{Straces}(\bar{s} \text{ after } \sigma \rho x) \). Now, to show that \( (\bar{s} \text{ after } \sigma \rho) \text{ios}(\bar{s} \text{ after } \sigma px) \), we only need to investigate if the first condition of Definition 6.34 holds for any \( \sigma' \in \text{Straces}(\bar{s} \text{ after } \sigma \rho x) \cap \text{Straces}(\bar{s} \text{ after } \sigma px) \). From our last observation, we obtain that \( \text{out}(\bar{s} \text{ after } \sigma \rho x \sigma') \subseteq U \cup \{\delta\} \subseteq \text{out}(\bar{s} \text{ after } \sigma \rho x \sigma') \), which was to be shown.

- We assume that \( \rho \notin \text{Straces}(\bar{s} \text{ after } \sigma x) \). Thus, we have to show that for any \( \sigma' \in \text{Straces}(\bar{s} \text{ after } \sigma px) \), the third condition is fulfilled, i.e., \( U \cup \{\delta\} \subseteq \text{out}(\bar{s} \text{ after } \sigma \rho x a x) \). We take \( \rho = \gamma b y' \) such that \( \gamma \in \text{Straces}(\bar{s} \text{ after } \sigma x) \) but \( \gamma b \notin \text{Sinit}(\bar{s} \text{ after } \sigma x) \) with \( b \in I \); thus the length of \( \gamma \) is less than \( n \). From the induction hypothesis, we obtain that \( x \in \text{out}(\bar{s} \text{ after } \sigma \gamma) \) and subsequently that \( (\bar{s} \text{ after } \sigma \gamma) \text{ios}(\bar{s} \text{ after } \sigma \gamma x) \).

Now, we distinguish between two cases; \( b \gamma' \in \text{Straces}(\bar{s} \text{ after } \sigma \gamma) \) and \( b \gamma' \notin \text{Straces}(\bar{s} \text{ after } \sigma \gamma) \).

  * We assume that \( b \gamma' \in \text{Straces}(\bar{s} \text{ after } \sigma \gamma) \). From Proposition 6.32, we deduce from \( b \notin \text{Sinit}(\bar{s} \text{ after } \sigma \gamma x) \) and \( b \in \text{Sinit}(\bar{s} \text{ after } \sigma \gamma x) \) that \( U \cup \{\delta\} \subseteq \text{out}(\bar{s} \text{ after } \sigma \gamma x b y' \gamma' \gamma' \gamma') \) for any \( \gamma' \in \text{Straces}(\bar{s} \text{ after } \sigma \gamma x b y' \gamma' \gamma') \). Using induction hypothesis another time, we conclude from \( \{x, b y' \} \subseteq \text{Sinit}(\bar{s} \text{ after } \sigma \gamma) \) that \( (\bar{s} \text{ after } \sigma \gamma x b y' \gamma' \gamma' \gamma') \text{ios}(\bar{s} \text{ after } \sigma \gamma b y' \gamma' x \gamma') \). Now, we distinguish three different cases for any \( \sigma' \in \text{Straces}(\bar{s} \text{ after } \sigma \gamma b y' \gamma' x) \).

    * We assume that \( \sigma' \in \text{Straces}(\bar{s} \text{ after } \sigma \gamma x b y' \gamma') \). Thus, we obtain from Definition 6.29 along with the two above results \( (\dagger) \) and \( (\ddagger) \) that \( \text{out}(\bar{s} \text{ after } \sigma \gamma x b y' \gamma' x a') = U \cup \{\delta\} \), which was to be shown.

    * We assume that \( \sigma' = \omega i \omega' \) with \( \omega, \omega' \in L_\omega \) and \( i \in I \) such that \( \omega \notin \text{Straces}(\bar{s} \text{ after } \sigma \gamma x b y' \gamma') \) while \( \omega i \notin \text{Straces}(\bar{s} \text{ after } \sigma \gamma x b y' \gamma') \).
Following Definition 6.29 along with the result (‡), we conclude that out(\(s\) after \(\sigma\gamma y\)' \(x\)' \(\sigma\)) = \(U \cup \{\delta\}\), which was to be shown.

- We assume that \(\sigma' = \omega_i\omega'\) with \(\omega,\omega' \in L^*_o\) and \(i \in U\) such that \(\omega \in \text{Straces}(\(s\) after \(\sigma\gamma x y\)') while \(\omega' \notin \text{Straces}(\(s\) after \(\sigma\gamma x y\)')\). Since \(\omega \in \text{Straces}(\(s\) after \(\sigma\gamma x y\)')\), we know from the result (‡) that out(\(s\) after \(\sigma\gamma x y\)') = \(U \cup \{\delta\}\). The last observation contradicts \(i \notin \text{out}(\(s\) after \(\sigma\gamma x y\)')\). Therefore, this case is not valid.

* We assume that \(y' \notin \text{Straces}(\(s\) after \(\sigma\gamma\)). From the induction hypothesis, we obtain from \{x, y'\} \(\subseteq\) Straces(\(s\) after \(\sigma\gamma\)) with \(x \in U\) and \(y' \in I^+\) that \(U \cup \{\delta\} \subseteq\) out(\(s\) after \(\sigma\gamma y'\)). Therefore, \(U \cup \{\delta\} \subseteq\) out(\(s\) after \(\sigma\rho x\)), which was to be shown.

\[
\square
\]

Now, we are in the position to establish the correctness of Theorem 6.36, which is the main result in this section.

**Proof of Theorem 6.36.** We have to show that \(\hat{\iota}\text{ioco}\hat{s}\) if and only if \(Q(\hat{\rho})\text{ioco}\hat{s}\). We divide the proof obligation into two parts: the right-to-left implication and vice versa. Due to \(\text{Straces}(\hat{\rho}) \subseteq \text{Straces}(Q(\hat{\rho}))\), it is not hard to verify that \(Q(\hat{\rho})\text{ioco}\hat{s}\) implies \(\hat{\iota}\text{ioco}\hat{s}\).

We therefore need to show the correctness of the right-to-left implication: \(\hat{\iota}\text{ioco}\hat{s}\) implies \(Q(\hat{\rho})\text{ioco}\hat{s}\). To this end, we prove the contrapositive version, i.e., \(Q(\hat{\rho})\text{ioco}\hat{s}\Rightarrow\hat{\iota}\text{ioco}\hat{s}\). It follows from \(Q(\hat{\rho})\text{ioco}\hat{s}\) that there exists a sequence \(\sigma \in \text{Straces}(\hat{s})\) such that out(\(Q(\hat{\rho})\) after \(\sigma\)) \(\not\subseteq\) out(\(s\) after \(\sigma\)); thus there exists an output \(x \in U \cup \{\delta\}\) such that \(x \in\) out(\(Q(\hat{\rho})\) after \(\sigma\)) but \(x \notin\) out(\(s\) after \(\sigma\)). We distinguish two cases; \(\sigma x \in \text{Straces}(\hat{\rho})\) and \(\sigma x \notin \text{Straces}(\hat{\rho})\). For the first case, i.e., \(\sigma x \in \text{Straces}(\hat{\rho})\), it is clear that \(\hat{\iota}\text{ioco}\hat{s}\). It therefore remains to show that the thesis holds for the latter case, i.e., \(\sigma x \notin \text{Straces}(\hat{\rho})\) implies \(\hat{\iota}\text{ioco}\hat{s}\). Thus, we assume that \(\sigma x \notin \text{Straces}(\hat{\rho})\). From Proposition 6.18, we know that there exists a sequence \(\sigma' \in \text{Straces}(\hat{\rho})\) such that \(\sigma'@\sigma x\). We continue the proof by induction on the number of delayed output actions in the sequence \(\sigma x\).

- **Base case.** We assume, for the induction basis, that the number of delayed output in \(\sigma x\) is one. We distinguish two cases; \(x\) is the only delayed output action in the sequence \(\sigma x\), and an output action \(y \neq x\) is delayed in that sequence.

  - We assume that \(x\) is the only delayed output action; thus \(\sigma' = \rho x\rho_i\) and \(\sigma = \rho\rho_i\) with \(\rho \in L^*_o, \rho_i \in I^+\). According to Definition 6.14, quiescence cannot be preceded by any input sequence that subsequently results in \(x \in U\). Towards a contradiction, we suppose that \(x \in\) out(\(\hat{s}\) after \(\rho\)); thus, \{x, \rho_i\} \(\subseteq\) Straces(\(\hat{s}\) after \(\rho\)) with \(x \in U\) and \(\rho_i \in I^+\). Since \(\hat{s}\) is a robust specification, we conclude from Proposition 6.37 that \(x \in\) out(\(\hat{s}\) after \(\rho\rho_i\)), which contradicts our assumption that \(x \notin\) out(\(\hat{s}\) after \(\rho\)). Therefore, we obtain that \(x \notin\) out(\(\hat{s}\) after \(\rho\)). We know that \(x \in\) out(\(\hat{\rho}\) after \(\rho\)), because \(\sigma' \in \text{Straces}(\hat{\rho})\). Combining the last two observations, we find that \(\hat{i}\text{ioco}\hat{s}\).

  - We assume that an output \(y \neq x\) is delayed in the sequence \(\sigma x\); thus, \(\sigma' = \rho y\rho_i\rho x\) and \(\sigma = \rho\rho_i\gamma\rho'\) for \(\rho, \rho' \in L^*_o, \rho_i \in I^+\). Like the previous case, we find that \(y \in U\). We first suppose that \(y \notin\) out(\(\hat{s}\) after \(\rho\)). Following \(\sigma' \in \text{Straces}(\hat{\rho})\), the last observation yields that \(\hat{\iota}\text{ioco}\hat{s}\). Next, we consider
the case that \( y \in \text{out}(s \text{ after } \rho) \); thus \( \{y, \rho_i\} \subseteq \text{Straces}(s \text{ after } \rho) \) with \( y \in U \) and \( \rho_i \in I^+ \). We investigate three different cases.

* We assume that \( \rho_i \rho' \in \text{Straces}(s \text{ after } \rho y) \). From Proposition 6.37, we obtain that \( (s \text{ after } \rho y \rho_i) \text{ios}(s \text{ after } \rho \rho_i y) \), and we consequently find that \( \text{out}(s \text{ after } \rho y \rho_i \rho') \subseteq \text{out}(s \text{ after } \rho \rho_i y \rho') \). We deduce from \( x \notin \text{out}(s \text{ after } \rho \rho_i y \rho') \) that \( x \notin \text{out}(s \text{ after } \rho y \rho_i \rho') \). Following \( x \in \text{out}(\tilde{r} \text{ after } \rho y \rho_i \rho') \), we obtain that \( \tilde{r} \ioco s \).

* We assume that \( \rho_i \rho' \notin \text{Straces}(s \text{ after } \rho y) \) because of an input, i.e., \( \rho_i \rho' = \gamma a' \gamma' \) such that \( a' \in I \) and \( \gamma \in \text{Straces}(s \text{ after } \rho y) \) but \( \gamma \rho \notin \text{Straces}(s \text{ after } \rho y) \). We distinguish three different cases based on the length of \( \gamma \).

First, we suppose that \( \gamma = \rho_i ; \) thus, \( \rho' = a \gamma' \). Following Proposition 6.37, we conclude that \( (s \text{ after } \rho y \gamma) \text{ios}(s \text{ after } \rho \gamma y) \). We deduce from Definition 6.29 that \( (s \text{ after } \rho \gamma y \gamma') = U \cup \{\delta\} \), which contradicts our assumption that \( x \notin \text{out}(s \text{ after } \rho \rho_i y \rho') \).

Second, we suppose that \( \gamma = \rho_i y'' \) with \( y'' \in L^* \); thus, \( \rho' = \gamma' a \gamma' \). From Proposition 6.37, \( (s \text{ after } \rho y \gamma y) \text{ios}(s \text{ after } \rho y \gamma y) \) is obtained. Subsequently, Proposition 6.32 yields \( (s \text{ after } \rho y \gamma y \gamma') = U \cup \{\delta\} \), which contradicts our assumption that \( x \notin \text{out}(s \text{ after } \rho \rho_i y \rho') \).

In the third case, we suppose that \( \rho_i = \gamma \rho_i y \) with \( \gamma_i \in I^+ \). Following Proposition 6.37, we obtain that \( (s \text{ after } \rho y \gamma y \gamma') = U \cup \{\delta\} \), which contradicts our assumption that \( x \notin \text{out}(s \text{ after } \rho \rho_i y \rho') \).

Therefore, by examining the above three cases, we find that the case where \( \rho_i \rho' \notin \text{Straces}(s \text{ after } \rho y) \) because of an input is not a valid one.

* We assume that \( \rho_i \rho' \notin \text{Straces}(s \text{ after } \rho y) \) because of an output; thus, \( \rho_i \rho' = \gamma z \gamma' \) such that \( z \in U \) and \( z \notin \text{out}(s \text{ after } \rho y \gamma) \). Since \( \rho_i \rho' \in \text{Straces}(\tilde{r} \text{ after } \rho y) \), we find that \( z \in \text{out}(\tilde{r} \text{ after } \rho y \gamma) \), which consequently results in \( \tilde{r} \ioco s \).

- Induction step. We assume that the thesis holds for all sequences with the number of delayed actions less than \( n \) and that the number of delayed actions in \( \sigma \) is \( n \). Thus, we take \( \sigma = \rho \rho_i y \rho' x \) where \( y \in U \) is proceeded by the sequence \( \rho_i \in I^+ \), and output \( y \) is the first delayed output in \( \sigma \), i.e., \( \sigma' = \rho y \rho' \) such that \( \sigma' \in \text{Straces}(\tilde{r}) \). First, we suppose that \( y \notin \text{out}(s \text{ after } \rho) \). Clearly, \( \tilde{r} \ioco s \), because \( y \in \text{out}(\tilde{r} \text{ after } \rho) \). Next, we consider the case where \( y \in \text{out}(s \text{ after } \rho) \). Before examining this case in details, we first show that \( Q(\tilde{r}) \text{ after } \rho y \rho_i = Q(\tilde{r}) \text{ after } \rho \rho_i y \). Since the input sequence \( \rho_i \) proceeds the output \( y \), we find that the sequence \( \rho \rho_i y \) is performed in the queue context as \( Q(\tilde{r}) \xrightarrow{\rho \rho_i y} [\rho_i < r' [\rho_i < r' \subseteq \text{straces}]] \xrightarrow{\gamma} [\rho_i < r' \subseteq \text{straces}] \) for some \( r', r'' \in R \) and \( \sigma_u \in U^* \). Following the inference rules of queue context, we obtain that the transition \( Q(\tilde{r}) \xrightarrow{\rho \rho_i y} [\rho_i < r' \subseteq \text{straces}] \xrightarrow{\rho \rho_i y} [\rho_i < r' \subseteq \text{straces}] \) is possible as well, which was to be shown. Now, we distinguish three cases.

  - We assume that \( \rho_i \rho'' \in \text{Straces}(s \text{ after } \rho y) \). Using the same line of reasoning as in the base case, we obtain that \( x \notin \text{out}(s \text{ after } \rho y \rho_i \rho'') \). We conclude from \( Q(\tilde{r}) \text{ after } \rho y \rho_i y = Q(\tilde{r}) \text{ after } \rho y \rho_i \rho' \) that \( x \in \text{out}(Q(\tilde{r}) \text{ after } \rho y \rho_i \rho'') \). The last two observations together results in \( \text{out}(Q(\tilde{r}) \text{ after } \rho y \rho_i \rho'') \notin \)
out(\bar{s} after \rho y\rho'|_{\rho''}) where the sequence \rho y\rho'|_{\rho''} has \( n - 1 \) delayed outputs. Following the induction hypothesis, we deduce from the last observation that \( \bar{r} \text{ioco} \bar{s} \).

- We assume that \( \rho_i\rho'' \not\in \text{Straces}(\bar{s} after \rho y) \) because of an input action, i.e., \( \rho_i\rho' = \gamma a'_\gamma' \) such that \( a'_i \in I \) and \( \gamma \in \text{Straces}(\bar{s} after \rho y) \) but \( \gamma a \not\in \text{Straces}(\bar{s} after \rho y) \). Analogous to the similar case in the base case, we find that this situation is not possible.

- We assume that \( \rho_i\rho'' \not\in \text{Straces}(\bar{s} after \rho y) \), because of an output action; thus, \( \rho_i\rho'' = \gamma z\gamma' \) such that \( z \in U \) and \( z \not\in \text{out}(\bar{s} after \rho y\gamma) \). Following \( Q(\bar{r}) \) after \( \rho_i\gamma = Q(\bar{r}) \) after \( \rho y\rho_i \), we obtain that \( \rho y\rho_i\rho'' \in \text{Straces}(Q(\bar{r})) \). Therefore, it is obvious that \( z \in \text{out}(Q(\bar{r}) after \rho y\gamma) \). Thus, we conclude that \( \text{out}(Q(\bar{r}) after \rho y\gamma) \not\subseteq \text{out}(\bar{s} after \rho y\gamma) \) for the sequence \( \rho y\gamma \) with \( n - 1 \) delayed output actions. From induction hypothesis, we obtain that \( \bar{r} \text{ioco} \bar{s} \).

\[ \square \]

We now establish the theorem below, which asserts that robustness of a specification is also a necessary condition to ensure the same verdict in the synchronous and the asynchronous settings, when standard ioco test cases are utilized.

**Theorem 6.38.** Let IOLTS \( \{S, I, U, \rightarrow, \bar{s}\} \) be a specification. Then, specification \( \bar{s} \) is robust if for any implementation \( \bar{r} \in \text{IOT}_S(I, U) \) we have \( \bar{r} \text{ioco} \bar{s} \) iff \( Q(\bar{r}) \text{ioco} \bar{s} \).

**Proof.** We prove the contrapositive version. Thus we assume that specification \( \bar{s} \) is not robust. It therefore suffices to show that there exists an implementation \( \bar{r} \) for which the predicate \( \bar{r} \text{ioco} \bar{s} \) if and only if \( Q(\bar{r}) \text{ioco} \bar{s} \) is false. So, we have to show that either \( \bar{r} \text{ioco} \bar{s} \Rightarrow Q(\bar{r}) \text{ioco} \bar{s} \) or \( Q(\bar{r}) \text{ioco} \bar{s} \Rightarrow \bar{r} \text{ioco} \bar{s} \) does not hold. Since \( \text{Straces}(\bar{r}) \subseteq \text{Straces}(Q(\bar{r})) \), we know that \( Q(\bar{r}) \text{ioco} \bar{s} \Rightarrow \bar{r} \text{ioco} \bar{s} \) holds for any specification and implementation. Therefore, we need to show that there exists an implementation \( \bar{r} \) such that \( \bar{r} \text{ioco} \bar{s} \) but \( Q(\bar{r}) \text{ioco} \bar{s} \). We distinguish three cases due to Definition 6.34.

- We assume that specification \( \bar{s} \) violates the first condition in Definition 6.34. Therefore, there exists a \( \sigma \in \text{Straces}(\bar{s}) \) such that \( \{x, a\} \subseteq \text{Straces}(\bar{s} after \sigma) \) with \( x \in U \), and \( a \in I \), whereas \( x \not\in \text{out}(\bar{s} after \sigma a) \). We suppose a conforming implementation \( \bar{r} \) such that \( \sigma \in \text{Straces}(\bar{r}) \) and \( x \not\in \text{out}(\bar{r} after \sigma) \). Therefore there exist states \( r' \) and \( r'' \) in the implementation such that \( \bar{r} \Rightarrow r' \Rightarrow r'' \). Following the inference rules in Definition 2.11, we know that there is transition \( Q(\bar{r}) \Rightarrow Q(r') \). As a direct consequence of deduction rules I3, A1, and A2, the transitions \( Q(r') \Rightarrow a \in r'' \Leftrightarrow x \Rightarrow \bar{r} \text{ioco} \bar{s} \).

- We assume that specification \( \bar{s} \) does not satisfy the second condition in Definition 6.34. Therefore, there exists a \( \sigma \in \text{Straces}(\bar{s}) \) such that \( \{x, a\} \subseteq \text{Straces}(\bar{s} after \sigma) \) with \( x \in U \), and \( a \in I \), whereas \( (\bar{s} after \sigma a) \text{ioco} \bar{s} \) after \( \sigma a x \), though \( a \in \text{Sinit}(\bar{s} after \sigma a x) \). From Definition 6.29, we know that there exists a sequence \( \sigma' \in \text{Straces}(\bar{s} after \sigma a x) \) such that one of the following conditions do not hold:

\( - \sigma \in \text{Straces}(\bar{s} after \sigma a x) \) implies \( \text{out}(\bar{s} after \sigma a x a' \sigma') \subseteq \text{out}(\bar{s} after \sigma a x a' \sigma') \)

127
We distinguish two cases based on which of the above conditions is not satisfied.

- We assume that the first condition is not fulfilled. Thus, there exists an output \( y \in U \cup \{ \delta \} \) such that \( y \notin \text{out}(\bar{s} \text{ after } \sigma x a) \) and \( y \notin \text{out}(\bar{s} \text{ after } \sigma x a') \). Now, we assume a conforming implementation \( \bar{r} \) of specification \( \bar{s} \) such that \( \sigma x a' \gamma' \in \text{Straces}(\bar{r}) \). We obtain from \( \text{Straces}(\bar{r}) \subseteq \text{Straces}(Q(\bar{r})) \), that \( \sigma x a' \gamma \in \text{Straces}(Q(\bar{r})) \); thus, \( y \in \text{out}(Q(\bar{r}) \text{ after } \sigma x a') \). From the last observation, along with \( Q(\bar{r}) \text{ after } \sigma x a' = Q(\bar{r}) \text{ after } \sigma x a' \), we find that \( y \in \text{out}(Q(\bar{r}) \text{ after } \sigma x a') \). Therefore, \( Q(\bar{r}) \text{ioco } \bar{s} \).

- We assume that the second condition is not fulfilled. Thus, there exist an input \( a \in I \) such that \( \sigma' = \gamma \gamma' \) and \( \gamma \in \text{Straces}(\bar{s} \text{ after } \sigma x) \) and \( \gamma a \notin \text{Straces}(\bar{s} \text{ after } \sigma x) \), but \( U \cup \{ \delta \} \not\subseteq \{ \bar{s} \text{ after } \sigma x a' \} \). Therefore, there exists an output \( y \in U \cup \{ \delta \} \) such that \( y \notin \text{out}(\bar{s} \text{ after } \sigma x a') \). We now take a conforming implementation \( \bar{r} \) such that \( \sigma x \gamma \gamma' \gamma \in \text{Straces}(\bar{r}) \). The rest of the proof is identical to the previous case.

- We assume that specification \( \bar{s} \) does not satisfy the second condition in Definition 6.34. Therefore, there exists a sequence \( \sigma \in \text{Straces}(\bar{s}) \) such that \( \{ x, a \} \subseteq \text{Straces}(\bar{s} \text{ after } \sigma) \) with \( x \in U \), and \( a \in I \), whereas \( a \notin \text{Sinit}(\bar{s} \text{ after } \sigma x) \) and \( U \cup \{ \delta \} \not\subseteq \text{out}(\bar{s} \text{ after } \sigma x a) \). Thus, there exist an output \( y \in U \cup \{ \delta \} \) such that \( y \notin \text{out}(\bar{s} \text{ after } \sigma x a) \). We take a conforming implementation \( \bar{r} \) such that \( \sigma x a y \in \text{Straces}(\bar{r}) \). The rest of the proof is similar to the second case of the previous item.

Combining the above two theorems, to be certain about the correctness of verdicts delivered by ioco-test cases even in the asynchronous setting, we just need to check if a given specification is robust. Since the ios relation is a generalization of the ioco coinductive relation, we can adapt the algorithm presented in Section 5.3.2 to assess the ios relation between states of the given specification at its race situations. This verification can be carried out prior to test case generation.

### 6.4 Closing Remarks

In this chapter, we studied the theoretical foundations for using the ioco testing framework in asynchronous conformance testing. To this end, we gave a full proof of the main theorem in [WW09; We09], where all classes of specifications, implementations, and test cases are confined to internal-choice IOLTSs. It was proven that the verdicts of internal-choice test cases are the same in both the synchronous and asynchronous settings, when both classes of implementations and specifications are restricted to internal-choice IOLTSs. It was also shown that this result cannot be utilized in the ioco testing framework: even if both classes of implementations and specifications are limited to internal-choice IOLTSs, the verdicts of internal-choice test cases may be different from those delivered by ioco test cases as illustrated in Example 6.13.
6.4. Closing Remarks

Subsequently, we presented theorems that allow for using test cases generated from ordinary specifications— not their composition with FIFO buffers— in order to test asynchronous systems. In this regard, we first introduced a subclass of IOTSs, namely delay right-closed IOTSs, as the class of implementations. We proved that in testing delay right-closed IOTSs the verdicts reached by \texttt{ioco} test cases in the asynchronous settings coincide with the verdicts delivered in the local testing through synchronous interaction. Afterwards, we showed that being delay right-closed for implementation is also the necessary condition to ensure the same verdict in the synchronous and the asynchronous settings.

The presented conditions on the IUT for synchronizing \texttt{ioco} are semantical in nature. Hence, the question naturally arises, what syntactic conditions imply the semantic conditions presented in this chapter, which are essential to make these theorems accessible in practice? For example, it is interesting to find out which composition of programming constructs and/or patterns of interaction satisfy the constraints established in this chapter on the class of implementations. Potentially interesting candidates are message-based multi-threaded systems that handle each receiving message in a separate thread. In such a system, because of the interleaving of process of messages from multiple threads, the set of observable executions of the system is closed with respect to the delay operator; thus a message-based multi-threaded system is a delay right-closed IOTS. The EFT switch of the case study reported in Chapter 3 is an example of a message-based multi-threaded system.

We also studied a subclass of specifications which ensures the correctness of verdicts delivered by \texttt{ioco}-test cases in testing ordinary implementations in the queue context. We proved that the imposed conditions on specifications are necessary and sufficient conditions. Using such a specification, we do not need to compose the models of asynchronous communication channels, \textit{i.e.}, FIFO queues, with the specification in testing when the IUT is accessible only via those kind of channels. In this way, we prevent the infamous state space explosion problem that is caused by composing queue models with the specification. Also, a large number of meaningless test cases that are generated because of queue models are removed. For example, by composing IOLTS $\delta$ in Figure 6.6 with two FIFO queues, sequence $p_r s p_r q p_r s r_r q$ is checked, which is an impossible execution; a purchase response is provided only in response to a purchase request. However, the important question raised is about the complexity of checking those presented conditions on ordinary IOLTSs. To this end, we need to study the complexity of the \texttt{ios} relation. By using our results in Chapter 5, we can show that checking the \texttt{ios} relation between IOLTSs has an exponential complexity in general. However, following the same approach in Chapter 5, we can find the polynomial time algorithm for deterministic IOLTSs in the size of their state spaces and transition relations.

The research reported in this chapter is inspired by the practical experience in testing an asynchronous system reported in Chapter 3. We still need to apply the insights obtained from this theoretical study to practical cases and find out to what extent the constraints of this chapter apply to either implementations or specifications in practice.
Enabling reuse and managing complexity are among the major benefits of using compositional approaches in software and systems engineering. This idea has been extensively adopted in several different subareas of software engineering, such as product-line software engineering. One of the cornerstones of the product-line approach is to reuse a common platform to build different products. This common platform should ideally comprise different types of artifacts, including test-cases, that can be re-used for various products of a given line. In this chapter, we propose an approach to conformance testing, which allows to use a high-level specification and derive specifications for to-be-developed components (or sub-systems) given the platform on which they are to be deployed. We call this approach decompositional testing and refer to the process of deriving specifications as quotienting (inspired by its counterpart in the domain of formal verification). We develop our approach within the context of ioco testing.

For a given platform (environment) $\bar{e}$, whose behavior is given as an IOLTS, a quotient of a specification $\bar{s}$ by the platform $\bar{e}$, denoted by $\bar{s}/\bar{e}$, is a specification that describes the system after filtering out the effect of $\bar{e}$. The structure of a system consisting of $\bar{e}$ and unknown component $\bar{c}$ is represented in Figure 7.1; the behavior of the system is described by a given specification $\bar{s}$. We would like to construct $\bar{s}/\bar{e}$ such that it captures the behavior of any component $\bar{c}$ which, when deployed on $\bar{e}$ (put in parallel and possibly synchronize with $\bar{e}$) conforms to $\bar{s}$. Put formally, $\bar{s}/\bar{e}$ is the specification which satisfies the following bi-implication:

$$\forall \bar{c}, \bar{e}. \ \bar{c} \ \text{ioco} \ \bar{s}/\bar{e} \ \Leftrightarrow \ \bar{c}||\bar{e} \ \text{ioco} \ \bar{s}$$

The criterion for the implication from left to right, which is essential for our approach, is called decomposability. The criterion for the implication from right to left guarantee that quotienting produces the precise specification for the component and is called strong decomposability. We study both criteria in this chapter. Moreover, we show that strong decomposability can be combined with on-the-fly testing, thereby avoiding constructing the witness to the decomposability explicitly upfront.
Chapter 7. Decomposability in Input Output Conformance Testing

The study of compositional and modular verification for various temporal and modal logics has attracted considerable attention and several compositional verification techniques have been proposed for such logics; see, e.g., [Par87; BCC98; KV97; PDH99; GPB05]. Decompositional reasoning aims at automatically decomposing the global property to be model checked into local properties of (possibly unknown) components, a technique that is often called quotienting. The notion of quotient introduced in the present paper is inspired by its corresponding notion in the area of (de)compositional model-checking, and is substantially adapted to the setting for input-output conformance testing, e.g., by catering for the distinction between input and output actions and taking care of (relative) quiescence of components. In the area of model-based testing, we are aware of a few studies dedicated to the issue of (de)composition [BRT03; FT06; Vil+12], of which we give an overview below.

In [BRT03] the compositionality of the ioco-based testing theory is investigated. Assuming that implementations of components conform to their specifications, the authors investigate whether the composition of these implementations still conforms to the composition of the specifications. They show that this is not necessarily the case and they establish conditions under which ioco is a compositional testing relation.

In [FT06], Frantzen and Tretmans study when successful integration of components by composing them in certain ways can be achieved. Successful integration is defined by two conditions: the integrated system correctly provides services, and interaction with other components is proper. For the former, a specification of the provided services of the component is assumed. Based on the ioco-relation, the authors introduce a new implementation relation called eco, which allows for checking whether a component conforms to its specification as well as whether it uses other components correctly. In addition, they also propose a bottom-up strategy for building an integrated system.

Another problem closely related to the problem we consider in this paper is testing in context, also known as embedded testing [Vil+12, Chapter 16]. In this setting, the system under test comprises a component c which is embedded in a context u. Component c is isolated from the environment and all its interactions proceed through u (which is assumed to be correctly implemented). The implementation i and specification s of the system composed of u and c, are assumed to be available. The problem of testing in context then entails generating a test suite that allows for detecting incorrect implementations i of component c.

Although testing in context and decomposability share many characteristics, there are key differences between the two. We do not restrict ourselves to embedded components,
nor do we assume the platforms to be fault-free. Contrary to the testing in context approach, decomposing a monolithic specification is the primary challenge in our work; testing in context already assumes the specification is the result of a composition of two specifications. Moreover, in testing in context, the component $\tilde{c}$ is tested through context $\tilde{u}$ whereas our approach allows for testing the component directly through its deduced specification. As a result, we do not require that the context is always available while testing the component, which is particularly important in case the platform is a costly resource.

For similar reasons, asynchronous testing studied in Chapter 6, which can be considered as some form of embedded testing, is different from the work we present in this paper.

**Structure of the chapter.** We study the basic definitions and concepts relevant to the theories developed in this chapter in Section 7.1. The notions of decomposability and strong decomposability are formalized in Section 7.2. We present sufficient conditions for determining whether a given specification is decomposable in Section 7.3 and whether it is strongly decomposable in Section 7.4. We conclude this chapter in Section 7.5.

### 7.1 Preliminaries

A software or hardware system is usually composed of subunits and modules that work in an orchestrated fashion to achieve the desired overall behavior of the software or hardware system. In Section 2.1.1 such compositions are formalized by using the parallel composition operator. However, all theories introduced in this thesis so far consider the IUT as a monolithic system. Our main goal in this chapter is to develop theories that pave the way for decomposing a given monolithic specification of a system with respect to its building modules. We study specifically the case where the IUT consists of two subsystems one of which is assumed to be well-known. Henceforth, we refer to the known subsystem and its counterpart within a compound system as the platform and unknown component respectively. We study in this chapter the problem that, how to derive the specification of the unknown component from the given specification and the platform.

In order to formally reason about the behaviors of the platform and the unknown component within the given specification, we require action alphabets for both the specification and the platform. Using the language of the platform, we are able to refine outputs of the whole system into two smaller sets: outputs of the platform and the unknown component respectively. The same refinement is also applicable on the set of input actions. Distinguishing between outputs of the building blocks of a system, we can refine the notion of quiescence to also incorporate stronger observations.

**Example 7.1.** The IOLTS $\hat{s}$ is a specification of a vending machine which sells tea for one euro coin ($\text{coin}$). After receiving money, it delivers $\text{tea}$ or it refunds the money ($\text{refund}$).

In practice, Vending machines often compromise at least two subsystems: a payment subsystem and a drink subsystem. The payment subsystem handles payment processes while the drink subsystem is responsible for serving the desired drink. Knowing the building modules of a vending machine, we as an external observer can distinguish between the
Chapter 7. Decomposability in Input Output Conformance Testing

Figure 7.2: A specification of a vending machine with inputs \{coin\} and outputs \{refund, tea\}

external behaviors of different parts of the machine. We naturally consider inserting a coin and receiving the money back (refund) as the inputs and the outputs of the payment module respectively, and subsequently, receiving a cup of tea (tea) as the output of the drink component. Distinguishing between the outputs of different modules in a compound system, we can even observe quiescence of them independently. For example, while the payment subsystem refunds the money, the drink component will not produce any drink and will stay quiescent.

For the purpose of the theory developed in this chapter, we refine some formal concepts introduced in Chapter 2, in the remainder of this section. Throughout this paper, we also confine ourselves to non-divergent IOLTS.

7.1.1 Quiescence and Suspension Traces

Differentiating between input and output actions, testers often not only have the power to observe events produced by an implementation, they can also observe the absence of events, or quiescence. In Definition 2.12 a quiescent state \( s \in S \), denoted by \( \delta(s) \), is defined as a state that does not produce outputs and is stable. In the same vein, we introduce relative quiescence of state \( s \) with regard to non-empty sets of output actions.

Let \( X \subseteq U \). We say that \( s \) is quiescent, relative to \( X \), denoted by \( \delta_X(s) \), whenever \( \text{init}(s) \subseteq L \setminus X \). Relative quiescence can intuitively be used to state that certain interfaces or communication channels are quiescent even if the overall implementation is capable of producing outputs over other channels or interfaces [Hee98]. Observe that \( \delta(s) \iff \delta_U(s) \); we hence sometimes use \( \delta(s) \) and \( \delta_U(s) \) interchangeably.

**Proposition 7.2.** Let \( \langle S, I, U, \rightarrow, \delta \rangle \) be an IOLTS, and let \( s \in S \). Let \( X, Y \subseteq U \) such that \( X \subseteq Y \). Then \( \delta_Y(s) \) implies \( \delta_X(s) \).

**Proof.** By definition of relative quiescence, it is concluded from \( \delta_Y(s) \) that \( \forall y \in Y \cdot y \notin \text{init}(s) \). Therefore, it follows from \( X \subseteq Y \) that \( \forall x \in X \cdot x \notin \text{init}(s) \). Hence, \( \delta_X(s) \).

In order to formally reason about the observation of relative quiescence, we introduce the set of extended suspension traces. To this end, we first generalize the transition relation \( \Rightarrow \) to a transition relation over extended suspension words. This relation is defined in the context of a non-empty collection \( \U \subseteq \mathcal{P}(U) \) of non-empty output sets. Let \( L_\U \) denote the set \( L \cup \delta_\U \), where \( \delta_\U \) is defined as the set \( \{ \delta_X \mid X \in \U \} \).

\[
\begin{align*}
\sigma s & \Rightarrow s \\
\sigma s & \Rightarrow_\U s \\
\delta_X(s) & \Rightarrow_X X \in \U
\end{align*}
\]
The following definition formalizes the set of extended suspension traces.

**Definition 7.3.** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOLTS, and let \( s' \in S \) and \( S' \subseteq S \). The set of extended suspension traces for \( s \) and \( S' \) are defined as follows:

- \( \text{Straces}_\mathfrak{U}(s) = \{ \sigma \in L^*_\mathfrak{U} \mid s \xrightarrow{\sigma} \mathfrak{U} \} \).

- \( \text{Straces}_\mathfrak{U}(S') = \bigcup_{s' \in S'} \text{Straces}_\mathfrak{U}(s') \).

We usually drop the subscript \( \mathfrak{U} \) if it only contains the set \( U \); we write \( \text{Straces}(S') \) rather than \( \text{Straces}_{\mathfrak{U}}(S') \). Note that our definition of \( \text{Straces}_{\mathfrak{U}}(S') \) coincides with the ordinary set of suspension traces \( \text{Straces}(S') \) defined in Definition 2.14.

Finally, we introduce the projection operator on extended suspension traces.

**Definition 7.4.** Let \( X \subseteq L \). The operator \( \llcorner \ : \mathcal{P}(L) \rightarrow L^*_\mathfrak{U} \rightarrow L^*_\mathfrak{U} \) is defined through the following rules:

\[
\epsilon_{\llcorner X} = \epsilon
\]

\[
(x\sigma)_{\llcorner X} = \begin{cases} x\sigma_{\llcorner X} & \text{if } x \in X \\ \sigma_{\llcorner X} & \text{otherwise} \end{cases}
\]

\[
(\delta_Y\sigma)_{\llcorner X} = \begin{cases} \delta_{XY\sigma} & \text{if } X \cap Y \in \mathfrak{U} \\ \sigma_{\llcorner X} & \text{otherwise} \end{cases}
\]

### 7.1.2 Input-Output Conformance Testing with Relative Quiescence

In this section, we extend the standard \text{ioco} relation over the extended suspension traces of a given specification. To this end, we first generalize the standard concepts that are used to define the \text{ioco} relation.

**Definition 7.5.** Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOLTS. Let \( s \in S \), \( S' \subseteq S \) and let \( \mathfrak{U} \subseteq \mathcal{P}(U) \) be a collection of non-empty output sets. Let \( \sigma \in L^*_\mathfrak{U} \).

- \( s \ \text{after}_\mathfrak{U} \sigma = \{ s' \in S \mid s \xrightarrow{\sigma} \mathfrak{U}, \sigma' \} \), and \( S' \ \text{after}_\mathfrak{U} \sigma = \bigcup_{s' \in S'} s' \ \text{after}_\mathfrak{U} \sigma \).

- \( \text{out}_\mathfrak{U}(s) = \{ x \in L^*_\mathfrak{U} \mid I \xrightarrow{x} s \} \), and \( \text{out}_\mathfrak{U}(S') = \bigcup_{s' \in S'} \text{out}(s') \).

Note that in case \( \mathfrak{U} = \{ U \} \), the above-defined notions coincide with the standard notions used to formalize the \text{ioco} theory, introduced in Section 2.3. The generalization of \text{ioco} that can take relative quiescence into account is formalized below.

**Definition 7.6.** Let \( \langle R, I, U, \rightarrow_r, \bar{r} \rangle \) be an IOTS representing a realization of a system, and let IOLTS \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be a specification. Let \( \mathfrak{U} \subseteq \mathcal{P}(U) \) be a collection of non-empty output sets, and let \( F \subseteq \text{Straces}_\mathfrak{U}(\bar{s}) \). We say that \( \bar{r} \ \text{input-output conforms} \) to specification \( \bar{s} \), denoted by \( \bar{r} \text{ ioco}_{\mathfrak{U}} \bar{s} \), iff

\[
\forall \sigma \in F : \text{out}_\mathfrak{U}(\text{after}_\mathfrak{U} \sigma) \subseteq \text{out}_\mathfrak{U}(\bar{s} \ \text{after}_\mathfrak{U} \sigma)
\]

The \text{ioco}_r conformance relation can be specialized by choosing an appropriate set \( F \). For instance, in a setting with \( \mathfrak{U} = \{ U \} \) and \( F = \text{Straces}(s) \), we obtain the standard \text{ioco} relation originally defined by Tretmans in [Tre96c].
7.1.3 Hiding actions

We can formalize communication between different components using the parallel operator \( \prod \) on IOLTSs: two IOLTSs can interact by connecting the outputs sent by one IOLTS to the inputs of the other IOLTS. We assume that such inputs and outputs are taken from a shared alphabet of actions. For the non-common actions, the behavior of both IOLTSs is interleaved. However, the interaction between components is unobservable by an external observer (a tester); the result of the communication is hidden. This is formalized below.

**Definition 7.7** (hiding). Let \( \langle S, I, U, \rightarrow, s \rangle \) be an IOLTS, and let \( V \subseteq U \). The IOLTS resulting from hiding actions in the set \( V \), denoted by \( \text{hide}[V] \) in \( s \) is the IOLTS \( \langle S, I, U \setminus V, \rightarrow, \vec{s} \rangle \), where \( \rightarrow' \) is defined as the least relation satisfying:

\[
\begin{align*}
\text{hide}[V] \text{ in } s \xrightarrow{1} \text{hide}[V] \text{ in } s' &\quad \text{if } x \notin V \\
\text{hide}[V] \text{ in } s \xrightarrow{x} \text{hide}[V] \text{ in } s' &\quad \text{if } x \in V
\end{align*}
\]

Note that successful communications over the alphabet of shared actions \( V \) are unobservable to a tester; such sequences are a potential source of divergence. As divergence is excluded from the ioco testing theory, we must assume such divergences in the parallel composition of two implementations do not take place. This implies that components that are put in parallel never produce infinite, uninterrupted runs of outputs over their alphabet of shared actions. Implementations adhering to these constraints are referred to as shared-output bounded implementations. Henceforth, we assume that implementations are shared-output bounded.

**Proposition 7.8.** Let \( \langle S, I, U, \rightarrow, s \rangle \) be an IOLTS. Let \( V \subseteq U \) and \( \Sigma \subseteq \mathcal{P}(U) \) such that \( V \) is a subset of all member of \( \Sigma \), i.e., \( \forall O \in \Sigma \bullet V \subseteq O \). We then have for all \( s \in S \) and all \( \sigma \in (L_U \setminus V)^* \):

\[
(\text{hide}[V] \text{ in } s) \ \text{after}_{\Sigma} \ \sigma = \{ (\text{hide}[V] \text{ in } s') | s' \in \vec{s} \ \text{after}_{\Sigma} \ \sigma' \wedge \sigma' \in L^*_U \wedge \sigma = \sigma'_{L(U \setminus V)} \}
\]

**Sketch of the proof.** We can divide the proof obligation into two parts: \( \forall \sigma \in (L_U)^* \bullet (\text{hide}[V] \text{ in } s) \ \text{after}_{\Sigma} \ \sigma \subseteq \{ (\text{hide}[V] \text{ in } s') | s' \in \vec{s} \ \text{after}_{\Sigma} \ \sigma' \wedge \sigma' \in L^*_U \wedge \sigma = \sigma'_{L(U \setminus V)} \} \) and \( \forall \sigma \in (L_U)^* \bullet \{ (\text{hide}[V] \text{ in } s') | s' \in \vec{s} \ \text{after}_{\Sigma} \ \sigma' \wedge \sigma' \in L^*_U \wedge \sigma = \sigma'_{L(U \setminus V)} \} \subseteq (\text{hide}[V] \text{ in } s) \ \text{after}_{\Sigma} \ \sigma \). Then, the proof of each of them proceeds by induction on the length of \( \sigma \).

7.1.4 Extended Suspension Automaton

The possible non-determinism that may be present in given specification \( \vec{s} \) and platform \( \vec{e} \) can cause complications in the decomposability problem that we study in this chapter. We largely avert this complication by utilizing the suspension automata underlying \( \vec{s} \) and \( \vec{e} \) (see Section 2.1.4). In [Tre96c], a transformation from ordinary IOLTSs to suspension automata is presented; the transformation ensures that trace-based testing using the resulting suspension automaton is exactly as powerful as ioco-based testing using the original IOLTS. Although converting an IOLTS to a suspension automaton preserves the observations over \( L \cup \{ \delta(\Sigma) \} \), it may obscure the observations over \( L_U \).

We solve this issue by extending standard suspension automata with a refined set of quiescence labels, reflecting the observations of quiescence induced by \( \Sigma \).
Definition 7.9 (Extended suspension automaton). An extended suspension automaton (ESA) is a deterministic LTS \( \langle S, L \cup \delta_{\mathcal{U}}, \rightarrow, \bar{s} \rangle \) where \( L \) is partitioned into two sets inputs \( I \) and outputs \( U \); that is, for all \( s \in S \) and all \( \sigma \in L_{\mathcal{U}}^* \), we have \( |s \text{ after}_{\mathcal{U}} \sigma| \leq 1 \).

Given an IOLTS and some collection \( \mathcal{U} \) of sets of output actions, the transformation \( \Delta \), defined below, converts any IOLTS into an ESA. The transformation is a straightforward generalization of the transformation to suspension automata presented in [Tre96c].

Definition 7.10 (Transformation function). Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOLTS over language \( L = I \cup U \). The ESA \( \Delta(\bar{s}) = \langle Q, I \cup U \cup \delta_{\mathcal{U}}, \rightarrow, \bar{q} \rangle \) is defined as follows:

- \( Q_\delta = \mathcal{P}(S) \setminus \{\emptyset\} \).
- \( q_0 = \bar{s} \text{ after}_{\mathcal{U}} \epsilon \).
- \( \rightarrow \subseteq Q \times L_{\mathcal{U}} \times Q \) is the least relation satisfying:

\[
\begin{align*}
    x & \in L & q & \in Q \\
    q \xrightarrow{x} \{s' \in S \mid \exists s \in q \cdot x \Rightarrow_{\mathcal{U}} s\} & q \xrightarrow{x} \{s \in q \mid \delta_X(s)\}
\end{align*}
\]

![Figure 7.3: Two suspension automata of a vending machine. Actions coin is an input action; actions refund and tea are outputs, and \( \delta \) and \( \delta_1 \) represent \( \delta_{\{\text{refund}, \text{tea}\}} \) and \( \delta_{\{\text{refund}\}} \) labels, respectively.](image)

Example 7.11. Consider the suspension automata depicted in Figure 7.3. The suspension automaton \( \bar{q} \) is derived by applying Tretmans’ original transformation [Tre96c] on IOLTS \( \bar{s} \) depicted in Figure 7.2. Note that the suspension traces of \( \bar{s} \) and the traces of suspension automaton \( \bar{q} \) are identical. Next, assume that \( \mathcal{U} = \{\{\text{refund}\}, U\} \). Then the extended suspension traces of \( \bar{s} \) differ from the traces of suspension automaton \( \bar{q} \): \( \delta_{\{\text{refund}\}} \), denoted by \( \delta_1 \) in Figure 7.3, is a possible observation in \( \bar{s} \), but it is not in \( \bar{q} \). Finally, consider extended suspension automaton \( \bar{p} \); ESA \( \bar{p} \) is derived from \( \bar{s} \); we also have \( \text{traces}(\bar{p}) = \text{Straces}_{\mathcal{U}}(\bar{s}) \).

Similar to an ordinary suspension automaton, an ESA may not generally represent an actual IOLTS. We therefore extend the notion of validity of a suspension automaton to an extended suspension automaton, see Section 2.1.4. This generalization is required for the proof of the last theorem in this chapter, but we believe it could be of interest in its own right in different contexts such as the automated learning context of [Wil06].
Definition 7.12. Let \( \langle Q, I \cup U \cup \delta_{\mathcal{U}}, \rightarrow, \bar{q} \rangle \) be an extended suspension automaton over language \( L = I \cup U \). We say that \( \bar{q} \) is valid iff it satisfies the following conditions:

- (non-blocking) for all \( q \in \text{der}(\bar{q}) \) and all \( X \in \mathcal{U} \), if \( \text{init}(q) \cap X = \emptyset \), then \( \delta_X \in \text{init}(q) \).
- (quiescent reducible) for all \( q \in \text{der}(\bar{q}) \), \( \sigma \in L^*_\mathcal{U} \) and \( X \in \mathcal{U} \), if \( \delta_X \sigma \in \text{traces}(q) \) then \( \sigma \in \text{traces}(q) \).
- (anomaly-free) for all \( q \in \text{der}(\bar{q}) \), \( X \in \mathcal{U} \) and \( x \in X \), we have \( \delta_X x \notin \text{traces}(q) \).
- (stable) for all \( q \in \text{der}(\bar{q}) \), there is some set \( \Sigma \subseteq L^*_\mathcal{U} \) such that for all \( \sigma \in \{\delta_Y \mid Y \in \mathcal{U}, Y \subseteq X\}^{*} \) and all \( q' \in Q \), if \( q \overset{\delta_x \sigma}{\Rightarrow} q' \) then \( \text{traces}(q') = \Sigma \).

The proposition below states that our transformation of IOLTSs to ESAs produces valid ESAs. This essentially proves that our notion of validity is not overly strict.

Proposition 7.13. Let \( \langle S, I, U, \rightarrow, \bar{s} \rangle \) be an IOLTS and let \( \mathcal{U} \) be a collection of output sets, i.e., \( \mathcal{U} \subseteq \mathcal{P}(U) \). Then \( \Delta(\bar{s}) \) is a valid extended suspension automaton.

Sketch of the proof. To show the correctness of the above thesis, we need to show that \( \Delta(s) \) fulfills all required conditions in Definition 7.12. Each property follows from the definition of transformation \( \Delta \).

Conversely, the theorem below states that any valid ESA has the same testing power (with respect to our generalization of \( \text{ioco} \)), as some IOLTS. This means that the class of valid ESAs can be used safely for testing purposes.

Theorem 7.14. Let \( \langle S, I \cup U \cup \delta_{\mathcal{U}}, \rightarrow, \bar{s} \rangle \) be a valid extended suspension automaton. Then, there is an IOLTS \( \langle S', I, U, \rightarrow, \bar{s}' \rangle \) over language \( L = I \cup U \), such that

- \( \text{traces}(\bar{s}) = \text{Straces}_{\mathcal{U}}(\bar{s}') \).
- for all \( F \subseteq \text{traces}(\bar{s}) \) and all implementations \( \bar{r}, \bar{r} \text{ioco}_{\mathcal{U}} \bar{s} \iff \bar{r} \text{ioco}_{\mathcal{U}} \bar{s}' \).

Sketch of the proof. Analogous to the proof of the corresponding theory of suspension automata in [Wil06], we first define the relation \( \sqsubseteq \) on state space \( S \) as \( s \sqsubseteq q \) iff \( \text{traces}(s) \subseteq \text{traces}(t) \). Subsequently, we define the equivalence relation \( \equiv \) on states as \( s \equiv t \) iff \( s \sqsubseteq t \) and \( t \sqsubseteq s \). We denote the \( \equiv \) equivalence class of \( s \) by \([s]_{\equiv}\). Now, consider IOLTS \( \langle S/_{\equiv}, I, U, \rightarrow, [\bar{s}]_{\equiv} \rangle \) with state-space \( S/_{\equiv} \) consisting of \( \equiv \) equivalence classes of \( S \) and the transition relation \( \rightarrow,_{\equiv} \) defined as follows:

- if \( x \in L \) and \( s \xrightarrow{x} t \), then \([s]_{\equiv} \xrightarrow{\epsilon}_{\equiv} [t]_{\equiv} \),
- if \( x \in \delta_{\mathcal{U}} \) and \( s \neq t \), then \([s]_{\equiv} \xrightarrow{x}_{\equiv} [t]_{\equiv} \).

IOLTS \([\bar{s}]_{\equiv}\) has the following properties:

1. \([s]_{\equiv} \xrightarrow{\epsilon}_{\equiv} [t]_{\equiv} \) implies \( t \sqsubseteq s \).
2. \( \forall s \in S \cdot \text{traces}(s) = \text{Straces}_{\mathcal{U}}([s]_{\equiv}) \).
3. \( \forall s \in S \cdot \forall \sigma \in L^*_{\mathcal{U}} \cdot \text{out}_{\mathcal{U}}(s \text{ after}_{\mathcal{U}} \sigma) = \text{out}_{\mathcal{U}}([s]_{\equiv} \text{ after}_{\mathcal{U}} \sigma) \).
We show that IOLTS \([\bar{s}]\) that is derived from ESA \(\bar{s}\) fulfills the two desired conditions. Following the construction of IOLTS \([\bar{s}]\), we find that it meets the first requirement. Thus, we need to show that \(\bar{r} \text{ioco}_{\bar{F}} \bar{s}\) iff \(\bar{r} \text{ioco}_{\bar{F}} [\bar{s}]\) for any IOTS \(\bar{r}\). Following the above properties, the proof of the latter statement follows using contraposition. □

We finish this section with lifting the parallel composition we defined for IOLTSs to the parallel composition on suspension automata, given below.

**Definition 7.15** (Parallel composition). Let \(\langle S_i, I_i \cup U_i \cup \{\delta\}, \rightarrow_i, \bar{s}_i \rangle\) for \(i = 1, 2\) be two suspension automata with disjoint input alphabets and disjoint output alphabets. The parallel composition of \(\bar{s}_1\) and \(\bar{s}_2\), denoted by \(\bar{s}_1 \parallel \bar{s}_2\), is an extended suspension automaton \(\langle Q, L \cup \delta_\|, \rightarrow, \bar{s}_1 \| \bar{s}_2 \rangle\), where:

- \(Q = \{s_1 \| s_2 \mid s_1 \in S_1, s_2 \in S_2\}\).
- \(L = I \cup U\) and \(\| = \{U_1 \cup U_2, U_1, U_2\}\), where \(I = (I_1 \cup I_2) \setminus (U_1 \cup U_2)\) and \(U = U_1 \cup U_2\).
- \(\rightarrow \subseteq Q \times (L \cup \delta_\|) \times Q\) is the least relation satisfying:

\[
\begin{align*}
\frac{s_1 \xrightarrow{x_1} s'_1 \quad x \notin L_2, x \neq \delta}{s_1 \| s_2 \xrightarrow{x} s'_1 \| s_2} & \quad \frac{s_2 \xrightarrow{x_2} s'_2 \quad x \notin L_1, x \neq \delta}{s_1 \| s_2 \xrightarrow{x} s'_2}
\end{align*}
\]

\[
\begin{align*}
\frac{s_1 \xrightarrow{\delta} s'_1}{s_1 \| s_2 \xrightarrow{\delta_{U_1}} s'_1 \| s_2} & \quad \frac{s_2 \xrightarrow{\delta} s'_2}{s_1 \| s_2 \xrightarrow{\delta_{U_2}} s'_1 \| s_2}
\end{align*}
\]

The result of composition of two suspension automata is, in general, an extended suspension automaton, which is valid, as the proposition below states.

**Proposition 7.16.** Let \(\bar{s}_1\) and \(\bar{s}_2\) be two suspension automata for which \(\bar{s}_1 \| \bar{s}_2\) is defined. Then \(\bar{s}_1 \| \bar{s}_2\) is a valid extended suspension automaton.

**Proof.** To show the correctness of the above proposition, we have to show that \(\bar{s}_1 \| \bar{s}_2\) meets the four requirements of a valid extended suspension automaton defined in Definition 7.12. Each property follows from the construction of \(\bar{s}_1 \| \bar{s}_2\). □

### 7.2 Decomposability

Software can be constructed by decomposing a specification of the software into sub-specifications of less complexity. Reuse of readily available, well-understood platforms or environments can steer such a decomposition. Given the prevalence of such platforms, the software engineering and associated testing problem thus shifts to finding a proper specification of the system from which the platform behavior has been factored out. Whether this is possible, however, depends on the specification; if so, we say that a specification is **decomposable**.

139
Chapter 7. Decomposability in Input Output Conformance Testing

The decomposability problem requires known action alphabets for both the specification and the platform. Hence, we first fix these alphabets and illustrate how these are related. Hereafter, \( L_s \) denotes the action alphabet of the specification \( \bar{s} \) and \( L_e \) denotes the action alphabet of the platform \( \bar{e} \). The actions of \( L_e \) not exposed to \( \bar{s} \) are contained in action alphabet \( L_v \), i.e., we have \( L_v = L_e \setminus L_s \). The action alphabet of the quotient is denoted by \( L \), i.e., \( L = (L_s \setminus L_e) \cup L_v \). The relation between the above alphabets is illustrated in Figure 7.1.

**Definition 7.17.** (Decomposability) Let IOLTS \( \bar{s} \) with actions alphabet \( L_s = I_s \cup U_s \) be a specification, and let IOTS \( \bar{e} \) with actions alphabet \( L_e = I_e \cup U_e \) be an implementation. Let \( L_v = I_v \cup U_v \) be the set of actions of \( \bar{e} \) not exposed in \( \bar{s} \), i.e., \( L_v = L_e \setminus L_s \). Specification \( \bar{s} \) is said to be decomposable for IOTS \( \bar{e} \) iff there is some IOLTS \( \bar{s}' \) with inputs \( I = (I_s \setminus I_e) \cup I_v \) and outputs \( U = (U_s \setminus U_e) \cup U_v \) for which:

\[
\forall \bar{c} \in \text{IOTS}(I, U) \bullet \bar{c} \text{ioco} \bar{s}' \Rightarrow (\text{hide}[L_v] \text{in} \bar{c} \parallel \text{ioco} \bar{s})
\]

Decomposability of a specification \( \bar{s} \) essentially ensures that a specification \( \bar{s}' \) for a sub-component exists that guarantees its every \text{ioco}-correct implementation works correctly in combination with the platform.

![Figure 7.4](image)

**Figure 7.4:** Two specifications of a drink component and an implemented money component with an implemented drink component building up a vending machine

**Example 7.18.** Consider IOLTSs depicted in Figure 7.4. IOTS \( \bar{e} \) presents the behavioral model of a money component implementation, which admits ordering a drink (\textit{order}) after receiving one euro coin (\textit{coin}). Subsequently, upon receiving error signal (\textit{error}), it refunds the money (\textit{refund}). The sets of input and output actions of component \( e \) are \( \{\text{coin, error}\} \) and \( \{\text{order, refund}\} \), respectively. Component \( e \) interacts with another component through actions \textit{order} and \textit{error}; together, the components implement a vending machine for which IOLTS \( \bar{s} \) in Figure 7.2 is the specification. IOLTS \( d \) is a specification of a drink component, which delivers \textit{tea} after receiving a drink \textit{order}. If it encounters a problem in delivering the drink, it signals an \textit{error}. Specification \( d \) guarantees that the
combination of component $e$ with any drink component implementation conforming to $d$, also conforms to IOLTS $\bar{s}$.

It may, however, be the case that there are implementations that, in combination with the given platform, adhere perfectly to the overall specification $\bar{s}$, and yet fail to pass the conformance test for $\bar{s}'$. As a consequence, non-conformance of an implementation to $\bar{s}'$ may not be sufficient reason to reject the implementation.

**Example 7.19.** Consider again IOLTSs depicted in Figure 7.4. IOLTS $d$ is a witness for decomposability of IOLTS $\bar{s}$ for platform $\bar{e}$. Thus, any compound system built of IOTS $\bar{e}$ and a component conforming to $\bar{m}$ is guaranteed to conform to specification $\bar{s}$. Consider IOTS $c'$, which incorrectly implements the functionality specified in IOLTS $d$, as it sends error twice. Observer, nevertheless, $\text{hide}[\{\text{error, order}\}]$ in $c' || e$ still conforms to $\bar{s}$.

It is often desirable to consider specifications $\bar{s}'$ for which one only has to check whether an implementation $\bar{c}$ adheres to $\bar{s}'$, i.e., specifications for which it is guaranteed that a failure of an implementation $\bar{c}$ to comply to $\bar{s}'$ also guarantees that the combination $\bar{c} || \bar{e}$ will violate the original specification $\bar{s}$. We can obtain this by considering a stronger notion of decomposability.

**Definition 7.20.** (Strong decomposability) Let IOLTS $\bar{s}$ with language $L_s = I_s \cup U_s$ be a specification, and let IOTS $\bar{e}$ with language $L_e = I_e \cup U_e$ be an implementation. Let $L_v = I_v \cup U_v$ be the set of actions of $\bar{e}$ not exposed in $\bar{s}$, i.e., $L_v = L_e \setminus L$. Specification $\bar{s}$ is said to be strongly decomposable for IOTS $\bar{e}$ iff there is some IOLTS $\bar{s}'$ with inputs $I = (I_s \setminus I_e) \cup I_v$ and outputs $U = (U_s \setminus U_e) \cup U_v$ for which:

$$\forall \bar{c} \in \text{IOTS}(I, U) : \bar{c} \text{ioco} \bar{s}' \iff \text{hide}[L_v] \text{in} \bar{c} || \bar{e} \text{ioco} \bar{s}$$

**Example 7.21.** Consider IOLTSs $d'$ and $e$ depicted in Figure 7.4. Specification $d'$ is such that the combination of component $\bar{e}$ with any shared output bounded component that does not conform to $d'$, fails to conform to $\bar{s}$.

### 7.3 Sufficient Conditions for Decomposability

A specification is decomposable with respect to a given platform, if there is a witness specification satisfying the required conditions defined in the previous chapter. However, knowing that a specification is decomposable in itself hardly helps a design engineer. Apart from the question whether a specification is decomposable, in practice, one is typically interested in a witness for the subspecification, or a quotient. Our approach to the decomposability problem is therefore constructive: we define a quotient and we identify several conditions that ensure that the quotient we define is a witness for the decomposability of a given specification.

One of the problems that may prevent a specification from being decomposable for a given platform $\bar{e}$ is that the latter may show some behavior which unavoidably violates the specification $\bar{s}$. We shall therefore only consider platforms without such violations. We formalize this by checking whether the behavior of $\bar{e}$ is included in the behavior of $\bar{s}$; that is, we give conditions that ensure that $\bar{e}$ in itself cannot violate the given specification.
Moreover, we assume that the input-enabled specification of \( e \) is available so that we can interchangeably reason about the implementation and specification of \( e \).

Assuming that the behavior of \( e \) is included in the behavior of the given specification \( s \), we then propose a quotient \( s' \) of \( s \) for \( e \) and prove sufficient conditions that guarantee that \( s \) is indeed decomposable and \( s' \) is a witness to that.

### 7.3.1 Inclusion Relation

We say that the behavior of a given platform is included in a specification if the outputs allowed by a specification subsume all outputs that can be produced by the platform, taking possible communications between the platform and the to-be-derived quotient over the action alphabet \( L_v \) into account. Another issue is that we are dealing with two components, each of which may be quiescent. If component \( e \) is quiescent, its quiescence may be masked by outputs from the component with which it is supposed to interact. We must therefore consider relative quiescence with respect to \( L_e \).

**Definition 7.22.** Let IOLTS \( \bar{s} \) with language \( L_s = I_s \cup U_s \) be a specification, and let IOTS \( \bar{e} \) with language \( L_e = I_e \cup U_e \) be an implementation. Let \( L_v = I_v \cup U_v \) be the set of actions of \( \bar{e} \) not exposed in \( \bar{s} \), i.e., \( L_v = L_e \setminus L \), and let \( \mathcal{U} = \{ U_e \setminus L_v, U_s \} \). We say the behavior of \( \bar{e} \) is included in \( \bar{s} \), denoted by \( \bar{e} \text{ incl} \bar{s} \) iff

\[
\forall \sigma \in \text{Straces}_\mathcal{U}(\bar{s}) : \text{out}(\text{hide}[L_v] \text{ in } \bar{e} \text{ after } \sigma_{|_{L_e}}) \subseteq \text{out}(\bar{s} \text{ after } \sigma)
\]

**Remark 7.23.** Every environment for which the specification does not describe any behavior, i.e., \( L_s \cap L_e = \emptyset \), is clearly included in the specification.

**Example 7.24.** Consider IOTS \( e \) in Figure 7.4 and IOLTSs \( \bar{s} \) depicted in Figure 7.2. The behavior of \( e \) is not included in \( \bar{s} \), because of the output "refund" in \( e \) after executing \((\text{coin tea})_{\bar{s}}\), while \( \bar{s} \) after execution of \((\text{coin tea})\) reaches a quiescent state. Next, consider IOTS \( r \) which is derived from IOTS \( e \) by replacing the "refund"-labeled transition in \( e \) by a \( \tau \)-labeled transition; IOLTS \( r \) has the same functionality of \( e \) except that upon receiving the error signal, it does not refund money. We have \( r \text{ incl} \bar{s} \).

### 7.3.2 Quotienting

We next focus on deriving a quotient of the specification \( \bar{s} \), factoring out the behavior of the platform \( \bar{e} \). To side step the complexity that arises as a result of the possible non-determinism in \( \bar{s} \) and \( \bar{e} \), we use the suspension automata underlying \( \bar{s} \) and \( \bar{e} \).

Another source of complexity is the fact that we must reason about the states of two systems running in parallel; such a system synchronizes on common actions and interleaves on non-common actions. We tame this conceptual complexity by formalizing an \texttt{executes} operator which, when executing a common or non-common action, keeps track of the set of reachable states for the (suspension automata) of \( \bar{s} \) and \( \bar{e} \). Formally, the \texttt{executes} operator is defined as follows. For the sake of simplicity and clarity in the remainder of this chapter, we implicitly show the absence of the outputs in specification \( \bar{s} \) by \( \delta_{\bar{s}} \) and similarly we use \( \delta_{\bar{e}} \) to denote quiescence relative to the outputs in platform \( \bar{e} \) (instead of using \( \delta_{\{U_s\}} \) and \( \delta_{\{U_e\}} \), respectively).
Definition 7.25. Let \( \langle Q_s, L_s \cup \{\delta_s\}, \rightarrow_s, \tilde{q}_s \rangle \) be a suspension automaton underlying specification IOLTS \( \tilde{s} \), and let \( \langle Q_e, L_e \cup \{\delta_e\}, \rightarrow_e, \tilde{q}_e \rangle \) be a suspension automaton underlying platform IOLTS \( \tilde{e} \). Let \( q \in \mathcal{P}(Q_s \times Q_e) \) be a non-empty collection of sets and let \( x \in (L_s \setminus L_e) \cup L_v \cup \{\delta_s\} \).

\[
q \text{ executes } x = \begin{cases}
\bigcup_{\sigma \in L_s} \bigcup_{(s,e) \in q} \{ (q'_s, q'_e) | s \xrightarrow{\sigma} q'_s \text{ and } e \xrightarrow{\sigma} q'_e \} & \text{if } x \in L_v \\
\bigcup_{\sigma \in L_s} \bigcup_{(s,e) \in q} \{ (q'_s, q'_e) | s \xrightarrow{\sigma} q'_s \text{ and } e \xrightarrow{\sigma} q'_e \} & \text{if } x \notin L_v \\
\bigcup_{\sigma \in L_s} \bigcup_{(s,e) \in q} \{ (q'_s, q'_e) | s \xrightarrow{\delta_s} q'_s \text{ and } e \xrightarrow{\delta_s} q'_e \} & \text{if } x = \delta_s
\end{cases}
\]

Using the executes operator, we have an elegant construction of an automaton, called a quotient automaton, see below, which allows us to define sufficient conditions for establishing the decomposability of a given specification.

Definition 7.26 (Quotient automaton). Let \( \langle Q_s, L_s \cup \{\delta_s\}, \rightarrow_s, \tilde{q}_s \rangle \) be a suspension automaton underlying specification \( \tilde{s} \), and let \( \langle Q_e, L_e \cup \{\delta_e\}, \rightarrow_e, \tilde{q}_e \rangle \) be a suspension automaton underlying platform \( \tilde{e} \). The quotient of \( \tilde{s} \) by \( \tilde{e} \), denoted by \( \tilde{s}_{/\tilde{e}} \), is a suspension automaton \( \langle Q, L \cup \{\delta\}, \rightarrow, \tilde{q} \rangle \), where:

- \( Q = (\mathcal{P}(Q_s \times Q_e) \setminus \{\emptyset\}) \cup Q_\delta \), where \( Q_\delta = \{ q_\delta \mid q \in \mathcal{P}(Q_s \times Q_e), q \neq \emptyset \} \); for \( q \notin Q_\delta \), we set \( q^{-1} = q \) and for \( q_\delta \in Q_\delta \), we set \( q_\delta^{-1} = q_\delta \).
- \( \tilde{q} = \{ (\tilde{q}_s, \tilde{q}_e) \} \).
- \( I = (I_s \setminus I_e) \cup (U_e \setminus U_s), U = (U_s \setminus U_e) \cup \{\delta\} \cup (I_e \setminus I_s), \) and \( L = I \cup U \).
- \( \rightarrow \subseteq Q \times L \times Q \) is the least set satisfying:

\[
\begin{array}{ll}
\frac{a \in I \quad q^{-1} \text{ executes } a \neq \emptyset \quad [I_1]}{q \xrightarrow{a} q^{-1} \text{ executes } a} & \frac{x \in U_v \quad q \notin Q_\delta \quad q^{-1} \text{ executes } x \neq \emptyset \quad [U_1]}{q \xrightarrow{x} q^{-1} \text{ executes } x} \\
\frac{x \in U_v \quad \forall (s,e) \in q, \sigma \in \text{traces}(s) \cap \text{traces}(e) : x \in \text{out}(s \text{ after } \sigma) \quad [U_2]}{q \xrightarrow{\sigma} q^{-1} \text{ executes } x} & \frac{\forall (s,e) \in q^{-1}, \sigma \in \text{traces}(s) \cap \text{traces}(e) : \delta_s \in \text{out}(s \text{ after } \sigma) \quad [\delta_1]}{q \xrightarrow{\delta_s} q^{-1} \text{ executes } \delta_s}
\end{array}
\]

We briefly explain the construction of a quotient automaton. A non-common input action is added to a state in the quotient automaton \( \tilde{s}_{/\tilde{e}} \), if an execution of the corresponding state in \( \tilde{e} \) leads to a state in \( \tilde{s} \) at which that action is enabled (\( I_1 \), in combination with the second case in Definition 7.25). A shared input action obeys the same rule except that a state of \( \tilde{e} \) has to be reachable where that input action is taken (\( I_1 \), in combination with the first case in Definition 7.25). Note that a shared input action of \( \tilde{s}_{/\tilde{e}} \) is an output action from the viewpoint of \( \tilde{e} \). In contrast, a non-shared output action is allowed at a state of \( \tilde{s}_{/\tilde{e}} \), only if it is allowed by \( \tilde{s} \) after any possible execution of \( \tilde{e} \) (\( U_2 \)) and a similar rule is applied.
to quiescence ($\delta_1$). Analogous to the shared input actions, a \textit{common output action} is considered as an action of a state whenever a valid execution of the corresponding states in $\bar{s}$ leads to a state at which that output action is enabled ($U_1$). Because the shared actions are hidden in $\bar{s}$, a shared output action, in $\bar{s}/\bar{e}$, may also be enabled at a state reached by $\delta$ transitions. Such a sequence of events is invalid due to the definition of quiescence. The observed problem is solved by adding a special set of states $Q_{\delta}$ to the states of the quotient automaton. These states represent quiescent states corresponding to the reachable states after executing $\delta$ in $\bar{s}/\bar{e}$. Moreover, no shared output action is added to these states.

![Figure 7.5](image)

\textbf{Figure 7.5:} A specification of a vending machine and a simple implemented money component with two quotient automata as proposed specifications for drink component

The quotient automaton derived from specification $\bar{s}$ and platform $\bar{e}$ is a suspension automaton: it is deterministic and it has explicit $\delta$ labels. Yet, the quotient automata we derive are not necessarily valid suspension automata. (As we recalled in Section 2.1.4, only valid suspension automata have the same testing power as ordinary IOLTSs.) We furthermore observe that there are some quotient automata that are valid suspension automata but nevertheless only admit non-shared output bounded implementations as implementations that conform to the quotient. Such implementations unavoidably give rise to divergent systems when composed in parallel with the platform.

\textbf{Example 7.27.} Consider IOLTSs and SAs depicted in Figure 7.5, and IOLTSs $\bar{s}$ in Figure 7.2. IOLTS $\bar{l}$ is a specification of a vending machine, which upon receiving a coin, either delivers tea, refund the coin, or does nothing. Similar to the money component $e$, depicted in Figure 7.4, IOTS $e'$ presents the behavioral model of an environment that after receiving a coin either orders a drink or does nothing. Upon receiving an error signal, it never refunds the coin. SA $\bar{r}$ is the quotient of $\bar{l}$ by $\bar{e}'$. Likewise, SA $\bar{i}$ is the quotient of $\bar{s}$ by $\bar{e}'$. Suspension automata $\bar{r}$ and $\bar{i}$ are valid SA regarding the definition of validity of suspension automata presented in [Wil06], See Definition 2.25. Assume an arbitrary shared output bounded IOTS $\bar{c}$ whose length of the longest sequence on the shared output is $n$, \textit{i.e.,}
out(\tilde{c} \text{ after } \sigma) \subseteq \{\text{tea}, \delta\} \text{ for } \sigma = \{\text{error}\}^n$. Clearly, \(\tilde{c} \text{ ioco } \tilde{t}\), because out(\tilde{t} \text{ after } \sigma) = \{\text{error}\}. However, for any \(n \geq 0\), there is always a shared output bounded IOTS that conforms to \(\tilde{r}\).

In view of the above, we say that a quotient automaton is valid if it is a valid suspension automaton and strongly non-blocking, given below.

**Definition 7.28.** Let \(\tilde{s}/\tilde{e}\) be a quotient automaton \(<Q, L \cup \{\delta\}, \rightarrow, \tilde{q}\>\) derived from a specification \(\tilde{s}\) and an environment \(\tilde{e}\). We say that \(\tilde{s}/\tilde{e}\) is valid iff both:

- \(\tilde{s}/\tilde{e}\) is a valid suspension automaton, and
- \(\tilde{s}/\tilde{e}\) is strongly non-blocking, i.e., \(\forall q \in Q \cdot \text{out}(q) \cap ((U \setminus U_s) \cup \{\delta\}) \neq \emptyset\).

Strongly non-blocking ensures that the quotient automaton always admits a shared output bounded implementation that conforms to it. Furthermore, valid quotient automata are, by definition, also valid suspension automata. Since every valid suspension automaton underlies at least one IOLTS, it therefore can be used in testing. Before the above claim is formally stated, the correctness of the following lemma, which establishes a link between traces of quotient automaton, the platform and the specification, is needed to be shown.

**Lemma 7.29.** Let IOLTS \(\tilde{s}\) be a specification, and let IOTS \(\tilde{e}\) be an environment. Let the behavior of IOTS \(\tilde{e}\) be included in the behavior of the given specification \(\tilde{s}\), i.e., \(\tilde{e} \text{ incl } \tilde{s}\), and let \(\sigma \in (L_e \cup L \cup \{\delta\})^\ast\) such that

- \(\sigma \downarrow \in \text{traces}(\tilde{s}/\tilde{e})\).
- \(\sigma \downarrow \in \text{Straces}(\tilde{e})\).
- \(\sigma\) ends in \((L \cup \{\delta\})\).

Then for all \(\sigma' \in \text{traces}(\tilde{s})\) satisfying \(\sigma' = \sigma \downarrow \text{ for } \tilde{e}\) we have:

\((\tilde{s} \text{ after } \sigma', \tilde{e} \text{ after } \sigma \downarrow) \in (\tilde{s}/\tilde{e} \text{ after } \sigma)\)^\text{-1}

**Proof.** The proof of the lemma follows using induction on the length of \(\sigma \downarrow\).

We are now in a position to establish the proposition below which states that a valid quotient automaton can be used in testing the to-be-developed component.

**Proposition 7.30.** Let IOLTS \(\tilde{s}\) be a specification, and let IOTS \(\tilde{e}\) be an environment, if

- \(\tilde{e} \text{ incl } \tilde{s}\)
- \(\tilde{s}/\tilde{e}\) is a valid quotient automaton

then,

\(\forall c \in \text{IOTS}(I, U) \cdot c \text{ ioco } \tilde{s}/\tilde{e} \implies (\text{hide}[L_s] \text{ in } c \parallel \tilde{e}) \text{ ioco } \tilde{s}\)

**Proof.** We will prove the contrapositive version of the above thesis, i.e., \(\forall c \in \text{IOTS}(I, U) \cdot (\text{hide}[L_s] \text{ in } c \parallel \tilde{e}) \text{ ioco } \tilde{s}\) implies \(c \text{ ioco } \tilde{s}/\tilde{e}\). Thus, we suppose that for some \(\sigma \in \text{Straces}(\tilde{s})\) and an output \(x \in (U_s \cup \{\delta_s\})\) we have,
• \( x \notin \text{out}(\tilde{s} \text{ after } \sigma) \)

• \( x \in \text{out}(\text{hide}[L_v]) \text{in} c \tilde{e} \text{ after } \sigma) \)

We now distinguish two cases: \( x = \delta_s \) and \( x \in U_s \). We then show that there exists a \( \sigma' \in (L_e U L \cup \{\delta_{(U U L_e)}\})^* \) such that \( \sigma'_i_{\bar{s}} = \sigma \) and also \( \delta \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \) for the case \( x = \delta_s \), or \( x \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \) otherwise; but \( x \notin \text{out}(\tilde{s}/\tilde{e} \text{ after } \sigma'_i_{\bar{s}}) \).

• We suppose that \( x = \delta_s \). Following \( x \in \text{out}(\text{hide}[L_v]) \text{in} c \tilde{e} \text{ after } \sigma) \), we find that there exists a state \( q \in (\text{hide}[L_v]) \text{in} c \tilde{e} \text{ after } \sigma) \) such that \( q \in (\text{hide}[L_v], \text{in} c \tilde{e}) \) is stable and does not produce any output of \( U_s \); thus, \( \text{init}(q) = (L \cup L_e) \setminus (L_v \cup U_s) \). We consequently deduce from stability of \( q \) in \( \text{hide}[L_v] \text{in} c \tilde{e} \) that \( \text{init}(q) \cap L_v = \emptyset \), because otherwise \( q \in \text{hide}[L_v] \text{in} c \tilde{e} \) would not be stable. Thus, \( \text{init}(q) \subseteq (L \cup L_e) \setminus (U_s \cup U_v) \).

Following Proposition 7.8, we know that there exists \( \sigma' \in (L_e L \cup \{\delta_{(U U L_e)}\})^* \) such that \( \sigma'_i_{\bar{s}} = \sigma \) and \( q \in (c \tilde{e} \text{ after } \sigma') \). Therefore, we obtain from (*) that \( \delta_{(U U L_e)} \in \text{out}(c \text{ after } \sigma') \). Regarding the definition of parallel composition (see Definition 2.5), we find that \( \delta \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \) and \( \delta \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \).

• We suppose that \( x \in U_s \). We know from \( x \in \text{out}(\text{hide}[L_v]) \text{in} c \tilde{e} \text{ after } \sigma) \) that there exists a state \( q \in (\text{hide}[L_v]) \text{in} c \tilde{e} \text{ after } \sigma) \) such that \( x \in \text{out}(q) \). Following Proposition 7.8, we know that there exists \( \sigma' \in (L_e L \cup \{\delta_{(U U L_e)}\})^* \) such that \( \sigma'_i_{\bar{s}} = \sigma \) and also \( q \in (c \tilde{e} \text{ after } \sigma') \); thus \( x \in \text{out}((c \tilde{e} \text{ after } \sigma')) \). Regarding the definition of parallel composition, we find from the last observation that \( x \in \text{out}(\tilde{e} \text{ after } \sigma'_i_{\bar{s}}) \) or \( x \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \). We proceed to show that \( x \in \text{out}(\tilde{e} \text{ after } \sigma'_i_{\bar{s}}) \) cannot be the case, because \( \tilde{e} \text{ incl } \tilde{s} \). We obtain that \( x \notin \text{out}(\text{hide}[L_v]) \text{in} c \tilde{e} \text{ after } \sigma'_i_{\bar{s}} \) combining \( \tilde{e} \text{ incl } \tilde{s} \) with \( x \notin \text{out}(\tilde{s} \text{ after } \sigma) \). Following \( \sigma = \sigma'_i_{\bar{s}} \), we consequently find that \( x \notin \text{out}(\text{hide}[L_v]) \text{in} e \text{ after } (\sigma'_i_{\bar{s}}) \).

Now, we continue the proof by distinguishing two cases with respect to \( \sigma'_i_{\bar{s}} \): \( \sigma'_i_{\bar{s}} \in \text{traces}(\tilde{s}/\tilde{e}) \) or otherwise. We show that in both cases, \( c \text{ioco } \tilde{s}/\tilde{e} \).

• We suppose that \( \sigma'_i_{\bar{s}} \in \text{traces}(\tilde{s}/\tilde{e}) \). We chop \( \sigma' \) into two parts, namely \( \rho' \) and \( \rho_{\tilde{e}} \) such that \( \rho' \) is the longest prefix of \( \sigma' \) that ends in \( L \cup \{\delta_{(U U L_e)}\} \); thus \( \rho_{\tilde{e}} \in (L_e \setminus L)^* \). We find \( \sigma = \rho'_i_{\bar{s}}, \rho_{\tilde{e}} \) due to \( \sigma = \sigma'_i_{\bar{s}} \), and also \( \sigma'_i_{\bar{s}} = \rho'_i_{\bar{s}} \). Following Lemma 7.29, we know that \( [(\tilde{s} \text{ after } \rho'_{\bar{s}}), (\tilde{e} \text{ after } \rho_{\tilde{e}})] \in (\tilde{s}/\tilde{e} \text{ after } \rho'_{\bar{s}})^{-1} \). We distinguish two cases: \( \delta \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \) or \( x \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \) with \( x \in U_s \).

- We suppose that \( x \in \text{out}(c \text{ after } \sigma'_i_{\bar{s}}) \) for \( x \in U_s \); thus, \( x \in U \setminus U_v \). We break this situation into two cases. First, we suppose that \( (\tilde{s}/\tilde{e} \text{ after } \rho'_{\bar{s}}) \notin \)
\[ \mathbb{P}(Q_x \times Q_y) \]. Therefore, \( \bar{s}/\bar{e} \) reaches a quiescent state that is reachable only by a \( \delta \)-labeled transition. Thus, the last step in \( \rho_{i_{\bar{e}}}^1 \) must be \( \delta \). Using \( \sigma'_{i_{\bar{e}}} = \rho_{i_{\bar{e}}} \), we consequently find that \( c \) after executing \( \sigma'_{i_{\bar{e}}} \) reaches quiescent states too; thus \( x \notin U \). So, this case is not possible.

Next, we suppose that \( (\bar{s}/\bar{e} \text{ after } \rho_{i_{\bar{e}}}^1) \in \mathbb{P}(Q_x \times Q_y) \). Combining inference rule \( U_2 \) along with \( x \notin \bar{s} \) after \( \rho_{i_{\bar{e}}}^1 \), we find that \( x \notin \text{out}(\bar{s}/\bar{e} \text{ after } \rho_{i_{\bar{e}}}^1) \).

Using \( \sigma'_{i_{\bar{e}}} = \rho_{i_{\bar{e}}} \), we obtain that \( c \text{ioco } \bar{s}/\bar{e} \) which was to be shown.

- We suppose that \( \delta \in \text{out}(c \text{ after } \sigma'_{i_{\bar{e}}} \) \). We know from Definition 7.26 that \( \delta \)-transitions in \( \bar{s}/\bar{e} \) are performed under deduction rule \( \delta_1 \). We find from \( \sigma'_{i_{\bar{e}}} \in \text{Straces}(\bar{e}) \) together with \( \sigma'_{i_{\bar{e}}} \in \text{Straces}(\bar{e}) \) that \( \rho_c \in \text{Straces}(\bar{e} \text{ after } \rho_{i_{\bar{e}}}^1) \). Combining the last two results, we obtain that \( \delta \notin \text{out}(\bar{s}/\bar{e} \text{ after } \rho_{i_{\bar{e}}}^1) \); thus, \( c \text{ioco } \bar{s}/\bar{e} \) which was to be shown.

- We suppose that \( \sigma'_{i_{\bar{e}}} \notin \text{traces}(\bar{s}/\bar{e}) \). We take \( \sigma' = \rho'z\sigma'' \) where \( z \in (L \cup \{\delta_1 \cup \bar{U} \cup U\}) \) and \( \rho', \sigma'' \in (L \cup L_e \cup (\delta_1 \cup \bar{U} \cup U))^* \) such that we have \( \rho_{i_{\bar{e}}}^1 \in \text{traces}(\bar{s}/\bar{e}) \) and \( z_{i_{\bar{e}}} \notin \text{init}(\bar{s}/\bar{e} \text{ after } \rho_{i_{\bar{e}}}^1) \). Therefore, there exists \( q' \in Q \cdot q \leftrightarrow q' \Rightarrow q' \not\rightarrow_\text{out} \) \( V_e \). We distinguish three cases based on \( z \).

  - We suppose that \( z = \delta_1 \cup \bar{U} \cup U \). We know that \( z_{i_{\bar{e}}} \in \text{out}(c \text{ after } \rho_{i_{\bar{e}}}^1) \) due to \( (\sigma')_{i_{\bar{e}}} \in \text{Straces}(c) \) with \( \sigma' = \rho'z\sigma'' \). We thus obtain from \( z_{i_{\bar{e}}} = \delta_1 \) that \( c \text{ioco } \bar{s}/\bar{e} \) which was to be shown.

  - We suppose that \( z \in (L \setminus \bar{U}) \). We break \( \rho' \) into two parts: \( \rho'' \) and \( \rho' \) such that \( \rho'' \) is the largest prefix of \( \rho' \) that ends in \( L \cup \{\delta_1 \cup \bar{U} \cup U\} \); thus \( \rho' \in (L_e \setminus L) \). We get from Lemma 7.29 that \( [(\bar{s} \text{ after } \rho''_{i_{\bar{e}}}^1), (\bar{e} \text{ after } \rho''_{i_{\bar{e}}}^1)] \in (\bar{s}/\bar{e} \text{ after } \rho''_{i_{\bar{e}}}^1)^{-1} \).

  - We suppose that \( z \in U \). Similar to the previous case, we break \( \rho' \) into two parts: \( \rho'' \) and \( \rho' \) such that \( \rho'' \) is the largest prefix of \( \rho' \) which ends in \( L \cup \{\delta_1 \cup \bar{U} \cup U\} \). We therefore find that \( \rho' \in (L_e \setminus L) \). We obtain from Lemma 7.29 that \( [(\bar{s} \text{ after } \rho''_{i_{\bar{e}}}^1), (\bar{e} \text{ after } \rho''_{i_{\bar{e}}}^1)] \in (\bar{s}/\bar{e} \text{ after } \rho''_{i_{\bar{e}}}^1)^{-1} \). Knowing that \( \rho''_{i_{\bar{e}}} \rho' \in \text{Straces}(\bar{e}) \) and \( \rho''_{i_{\bar{e}}} \rho' \in \text{Straces}(\bar{e}) \), we find in the same line of reasoning in the previous case that \( z \notin I \). We then distinguish two cases. First, we suppose that \( q' \in \mathbb{P}(Q_x \times Q_y) \); thus \( q' = q' = \bar{s}/\bar{e} \text{ after } \rho''_{i_{\bar{e}}}^1 \).

Second, we suppose that \( q' \notin \mathbb{P}(Q_x \times Q_y) \); thus \( q' \neq q' \). Therefore, \( q' \) is a quiescent state and it is reachable only by \( \delta \)-labeled transition; thus, the last step in \( \rho''_{i_{\bar{e}}}^1 \) must be \( \delta \). We therefore conclude that \( z \notin U_v \) because \( c \) after executing \( \rho''_{i_{\bar{e}}}^1 \) reaches quiescent states where no output is produced. Hence, we show through the above reasoning that, in general, \( z \notin U_v \). Therefore, \( z \in L_v \) cannot be the case.

147

7.3. Sufficient Conditions for Decomposability
The above proposition guarantees that a composite system built up of a conforming implementation of valid \( \bar{s}/\bar{e} \) and platform \( \bar{e} \) also conforms to specification \( \bar{s} \). We establish in the theorem below a sufficient condition for the decomposability of a specification.

**Theorem 7.31.** Let IOLTS \( \bar{s} \) be a specification, and let IOTS \( \bar{e} \) be an environment. Then \( \bar{s} \) is decomposable for \( \bar{e} \) if \( \bar{s}/\bar{e} \) is a valid quotient automaton and \( \bar{e} \) incl \( \bar{s} \).

**Proof.** It is the direct result of Proposition 7.30 and definition of decomposability (see Definition 7.17).

Note that the IOLTS underlying the quotient automaton is a witness to the decomposability of the specification; we thus not only have a sufficient condition for the decomposability of a specification but also a witness for the decomposition.

### 7.3.3 Example

The various involving parties in every transaction performed by an EFT switch, which we studied in Chapter 3, in conjunction with the variety of financial transactions, complicate the behavioral model of the EFT switch. Similar to any other complex software system, the EFT switch comprises many different components, some of which can be run individually. So, to illustrate the notions introduced so far, we treat a simplified model of purchase transaction in an EFT.

A part of the simplified communication model of the EFT switch with a banking system in the purchase scenario is depicted in Figure 7.6(a). The scenario starts by receiving a purchase request from a POS; this initial part of the scenario is removed from the model, for the sake of brevity. Subsequently, the EFT switch sends a purchase request (p_rq) to the banking system. The EFT switch will reverse (rev_rq) the sent purchase request if the corresponding response (p_rs) is not received within a certain amount of time (e.g., an internal time-out occurs, denoted by \( \tau \)). Due to possible delays in the network layer of the EFT switch, an external observer (tester) may observe the reverse request of a purchase even before the purchase request which is pictured in Fig 7.6(a).

The EFT switch is further implemented in terms of two components, namely, the financial component and the reversal component. A simplified behavioral model of the financial component is given in Figure 7.6(b). Comparing the two languages of \( \bar{s} \) and \( \bar{e} \), action \( t \) (representing time-out) is part of an internal interface between \( \bar{e} \) and a to-be-developed implementation of the reversal component. We observe that the behavior of \( \bar{e} \) is included in the behavior of specification \( \bar{s} \), because for every sequence \( \sigma \) in:

\[
(p_{-rq}(\delta_{\bar{e}}[\text{rev}_{-rq}]^\ast,p_{-rq}(\delta_{\bar{e}}[\text{rev}_{-rq}]^\ast\text{rev}_{-rq}(\delta|\delta_{\bar{e}})^\ast,p_{-rq}p_{-rs}(\delta_{\bar{e}}[\text{rev}_{-rq}]^\ast(\delta|\delta_{\bar{e}})^\ast, (\delta_{\bar{e}}[\text{rev}_{-rq}]^\ast,(\delta_{\bar{e}}[\text{rev}_{-rq}]^\ast\text{rev}_{-rq}(\text{rev}_{-rq}|p_{-rq})^\ast(\delta|\delta_{\bar{e}})^\ast, \text{out}(\text{hide}[t]\text{in }\bar{e} \text{ after } \sigma_{\bar{e}})) \subseteq \text{out}(\bar{s} \text{ after } \sigma) \text{ always holds.}
\]

We next investigate the decomposability of \( \bar{s} \) with \( \bar{e} \), by constructing the quotient \( \bar{s}/\bar{e} \). Note that \( t \) is the only shared action which is an input action from the viewpoint of \( \bar{s}/\bar{e} \). The resulting quotient automaton, obtained by applying Definition 7.26 to \( \bar{s} \) and \( \bar{e} \) is depicted in Figure 7.6(c). We illustrate some steps in its derivation. The initial state of the quotient automaton is defined as \( \{(\bar{s}, \bar{e})\} \). Below, we
7.3. Sufficient Conditions for Decomposability

Figure 7.6: A schematic view of the EFT Switch, a suspension automata of simplified behavioral models of the EFT switch \( \hat{s} \) and an implementation of the financial component \( \hat{e} \), and the quotient of \( \hat{s} \) w.r.t. \( \hat{e} \)

Illustrate the rules of Definition 7.26 that are possible from this initial state; doing so repeatedly for all reached states will ultimately produce the reachable states of the quotient automaton.

1. We check the possibility of adding input transitions to \( q_0 = \{(\hat{s}, \hat{e})\} \). Following \( q_0 \) executes \( t = \{(s_1, e_2)\} \) and deduction rule \( I_1 \) in Definition 7.26, the transition \( q_0 \xrightarrow{t} q_1 \) is added to the transition relation of \( \hat{s}/\hat{e} \), where \( q_1 = \{(s_1, e_2)\} \).

2. We check the possibility of adding output transitions to \( q_0 = \{(\hat{s}, \hat{e})\} \). We observe that \( \text{rev}_rq \in \text{out}(\hat{s} \text{ after } \sigma) \) for every \( \sigma \in \{\epsilon, p_rq, p_rq p_rs\} \). Regarding deduction rule \( U_2 \), the transition \( q_0 \xrightarrow{\text{rev}_rq} q_2 \) is added to the transition relation of \( \hat{s}/\hat{e} \), where \( q_2 = \{(s_5, \hat{e}), (s_2, e_1), (s_2, e_3)\} \).

3. Following deduction rule \( \delta_1 \) and \( \delta \notin \text{out}(\hat{s} \text{ after } \epsilon) \), the \( \delta \)-labeled transition is not added to \( q_0 \).

The constructed quotient automaton \( \hat{s}/\hat{e} \) is valid: it is both a valid suspension automaton and strongly non-blocking. As a result, \( \hat{s} \) is decomposable with respect to \( \hat{e} \) and \( \hat{s}/\hat{e} \) is a witness to that.
7.4 A Condition for Strong Decomposability

It is a natural question whether the quotient automaton that we defined in the previous section, along with the sufficient conditions for decomposability of a specification are also sufficient conditions for strong decomposability. The proof of Proposition 7.30 gives some clues to the contrary. The main problem is in the notion of quiescence, and, in particular in the notion of relative quiescence, which is unobservable in standard io-co theory. More specifically, the platform ε may mask the (unwanted) lack of outputs of the quotient automaton.

A natural solution to this is to confine platform ε to IOTS ⊓ s, introduced in Section 2.1.3: such implementations only accept inputs when reaching a quiescent state. A quiescent state, therefore, is always reachable after any execution in ε. Thus, platform ε behaving as an IOTS ⊓ ensures that it never masks quiescence of other components in their composition.

The following lemma establishes a link between the traces of a quotient automaton and the suspension traces of the specification when the given platform is assumed to behave as an IOTS ⊓.

Lemma 7.32. Let IOLTS ¯s be a specification, and let IOTS ⊓ ¯e be an environment, if

• ¯e incl ¯s
• ¯s/¯e is a valid quotient automaton (with initial state ¯q)

then, for every σ ∈ traces(¯s/¯e), and every (s, e) ∈ q−1 with q = ¯q after σ, we also have,

∃σ′ ∈ (L ∪ L_e ∪ {δ(U ∪ U_e)})* • s = (Δ(¯s) after σ′_s) ∧ (q, e) = (¯q∥Δ(¯e) after σ′)

Proof. For this correspondence, we rely on the fact that ¯q∥Δ(¯e) is a valid ESA (see Proposition 7.16 ). The proof of the lemma then follows using induction on the length of σ. □

Now, we can establish the proposition below which states that the quotient automaton captures exactly the behavior of any shared output bounded component that combines with platform ε behaving as an IOTS ⊓, conforms to specification ¯s.

Proposition 7.33. Let IOLTS ¯s be a specification, and let IOTS ⊓ ¯e be an environment, if

• ¯e incl ¯s, and
• ¯s/¯e is a valid quotient automaton

then

∀c ∈ IOTS(I, U) • hide[L_v] in c∥io-co ¯s implies c io-co ¯s/¯e

Proof. We prove the contrapositive version of the above thesis, i.e., ∀c ∈ IOTS(I, U) • c io-co ¯s/¯e ⇒ hide[L_v] in c∥io-co ¯s. Thus, we assume that there exists sequence σ ∈ traces(¯s/¯e) such that for some x ∈ U ∪ {δ}

• x /∈ out(¯s/¯e after σ)

150
7.4. A Condition for Strong Decomposability

- $x \in \text{out}(c \text{ after } \sigma)$

Since $x \in \text{out}(c \text{ after } \sigma)$, we know that there exists $c' \in c \text{ after } \sigma$ such that $x \in \text{out}(c' \text{ after } e)$. For the sake of simplicity, we call the reached state in $\delta(e)$ after executing $\sigma$, $q$, i.e., $q = \bar{q}$ after $\sigma$, and also we use $\delta(e)$ and $\bar{\delta}(e)$ to denote $\Delta(\bar{s})$ and $\Delta(\bar{e})$ respectively. We distinguish two cases based on $q^{-1}$. First, we assume that $q^{-1} \neq q$; thus, $q \notin \mathbb{P}(Q_s \times Q_e)$. Following Definition 7.26, we know that $q$ is reachable only by $\delta$-labeled transitions; thus, the last step in $\sigma$ must be $\delta$. We consequently find that $c$ after executing $\sigma$ reaches to quiescent state. Thus, $c'$ is quiescent and it subsequently produces no output. The last two observations result in $\text{out}(c' \text{ after } e) = \text{out}(q) = \{\delta\}$, which contradicts the assumption that $\bar{q}$ after executing $\sigma$ produces a different output than $c$ after $\sigma$. Thus, $q^{-1} \neq q'$ cannot be the case. Second, we assume that $q^{-1} = q$; thus, $q \in \mathbb{P}(Q_s \times Q_e)$. We distinguish three cases regarding $x$.

- We suppose that $x \in U_v$. Since environment $\bar{e}$ behaves as an IOTS$^-$, we conclude from $U_v \subseteq I_e$ that for every $\sigma_e \in (L_e \cup \{\delta_e\})^*$ it always holds $x \in \text{init}(\bar{e}_\delta \text{ after } \sigma)$. Following deduction Rule $U_1$ in Definition 7.26, we thus obtain that $x \in \text{init}(q)$; thus, this case is not possible.

- We suppose that $x \in (U \setminus U_v)$. Regarding inference rule $U_2$, we deduce from $x \notin \text{out}(q)$ that there exists a pair $(s, e) \in q^{-1}$ such that for some $\sigma_e \in \text{traces}(s) \cap \text{traces}(e)$, we have that $x \notin \text{out}(s \text{ after } \sigma_e)$. Following Lemma 7.32, we know that for the pair $(s, e)$ there exists $\sigma' \in (L \cup L_e \cup \{\delta_{(U \cup U_1)}\})^*$ such that $s = \delta_s \text{ after } \sigma'_i$ and also, $q|e = q|\bar{e}_\delta \text{ after } \sigma'$. We find from the last two observations that $x \notin \text{out}(\delta_s \text{ after } \sigma'_i, \sigma_e)$ (†). Furthermore, We obtain from the definition of parallel composition that $e = \bar{e}_\delta \text{ after } \sigma'_i$. Similarly, $q = q \text{ after } \sigma'_i$; thus, $\sigma'_i = \sigma'$. Combining the last observation with $\sigma_e \in \text{traces}(e)$, we have $\sigma'_i, \sigma_e \in \text{traces}(\bar{e}_\delta)$. Therefore, there exist $e', e''$ such that $\bar{e} \sigma'_i \sigma_e \Rightarrow e' \sigma e''$. Following the definition of parallel composition, we have the transition $c|e \Rightarrow \sigma_e c|e' \Rightarrow c'|e''$. Using $x \in \text{out}(c' \text{ after } e)$, we find that $x \in \text{out}(c'|e'')$. From Proposition 7.8, we find that $c'|e'' \Rightarrow \text{hide}([L_v]) c|\bar{e} \text{ after } (\sigma'_{\sigma_e})_{i_1}$. We consequently obtain that $x \in \text{out}([L_v]) c|\bar{e} \text{ after } (\sigma'_{\sigma_e})_{i_1}$. Regarding the last observation with (†), we deduce that $\text{out}((\text{hide}([L_v]) c|\bar{e} \text{ after } (\sigma'_{\sigma_e})_{i_1})) \notin \text{out}(\delta_s \text{ after } (\sigma'_{\sigma_e})_{i_1})$. Consequently, we obtain from Theorem 7.14 together with the last observation that $\text{out}((\text{hide}([L_v]) c|\bar{e} \text{ after } (\sigma'_{\sigma_e})_{i_1})) \notin \text{out}(\delta_{(U \cup U_1)} \text{ after } (\sigma'_{\sigma_e})_{i_1})$. We therefore find that $\text{hide}([L_v]) c|\bar{e}$ \text{iqco} $\delta$, which was to be shown.

- We suppose that $x = \delta$. Following inference rule $\delta_1$, we know that there exists a pair $(s, e) \in q$ such that for some $\sigma_e \in \text{traces}(s) \cap \text{traces}(e)$, it holds that $\delta_s \notin \text{out}(s \text{ after } \sigma_e)$. We find from Lemma 7.32 that there exists $\sigma' \in (L \cup L_e \cup \{\delta_{(U \cup U_1)}\})^*$ such that $s = \delta_s \text{ after } \sigma'_i$ and also, $q|e = q|\bar{e}_\delta \text{ after } \sigma'$. The last two observations result in $\delta_s \notin \text{out}(\delta_s \text{ after } \sigma'_i \sigma_e)$ (∗). Moreover, using the same line of reasoning as in the previous case, we find that there exists a state $e'$ in $\bar{e}$ such that $c'|e' \in c|\bar{e} \text{ after } \sigma'_i \sigma_e$. We know that, since $\bar{e}$ behaves as an IOTS$^-$ that after executing any trace, $\bar{e}$ can always reach a quiescent state. Therefore, $\delta_{(U \cup U_1)} \in \text{out}(c'|e' \text{ after } e)$. Following Proposition 7.2, we obtain from $U_s \subseteq$
Chapter 7. Decomposability in Input Output Conformance Testing

Let IOLTS $\bar{s}$ be a specification and let IOTS $\bar{e}$ be an environment. If $\bar{s}$ is decomposable, then

$$\forall c \in IO TS(I, U) : c \, ioco \bar{s} / \bar{e} \iff hide[L_v] \text{in} c \parallel \bar{e} \, ioco \bar{s}$$

Proof. The correctness of the above thesis directly follows from Theorem 7.31 and Proposition 7.33. $\square$

As a result of the above theorem, testing whether the composition of a component $\bar{c}$ and a platform $\bar{e}$ conforms to specification $\bar{s}$ reduces to testing for the conformance of $\bar{c}$ to $\bar{s} / \bar{e}$. This can be done using the standard ioco testing framework [Tre08].

A problem may arise when trying this approach in practice. Namely, the amount of time and memory needed for derivation of the ioco test suit increases exponentially in the number of transitions in the specification due to the nondeterministic nature of the test case generation algorithm. We avoid these complexities by presenting an on-the-fly testing algorithm inspired by [VT00]. Algorithm 7.4.1 describes the on-the-fly testing algorithm in which sound test cases are generated without constructing the quotient automaton upfront. We partially explore the quotient automaton during test execution. We use an extended version of the executes operator in Algorithm 7.4.1 which is defined on ordinary IOLTSs; the underlying IOLTSs of suspension automata are used to avoid the complexity of constructing suspension automata, i.e., $executes : P(P(S_i) \times P(S_j)) \times L_{\bar{s}} \times P(P(S_i) \times P(S_j))$.

Termination of the above algorithm with $V = \text{Fail}$ implies that the composition of the implementation under test with $\bar{e}$ does not conform to $\bar{s}$.

Theorem 7.35. Let IOLTS $\bar{s}$ be a specification and let IOTS $\bar{e}$ be an environment whose behavior is included in $\bar{s}$, i.e., $\bar{e} \, incl \bar{s}$. Let $V$ be the verdict upon termination of Algorithm 7.4.1 when executed on an implementation $\bar{c}$. If $hide[L_v]\text{in} \bar{e} \parallel \bar{c} \, ioco \bar{s}$ then $V = \text{Pass}$.

Proof. We prove the contrapositive version of the above thesis, i.e., $\forall \bar{c} \in IO TS(I, U) : \bar{c} \, \text{fails } t \implies hide[L_v] \text{in} \bar{c} \parallel \bar{e} \, ioco \bar{s}$.

We assume $\sigma \in L_{\bar{s}}$ is the sequence whose execution brings $t$ to fail state. Therefore, there exists $x \in U \cup \{\delta\}$ such that $x \in \text{out}(\bar{c} \after \sigma)$ but $t$ fails upon observing it. Regarding Algorithm 7.4.1, we obtain that $x \notin \text{out}(\bar{s} \after \sigma)$. Then, the remainder of the proof proceeds in the same lines of reasoning of the proof of Proposition 7.33. $\square$
Algorithm 7.4.1: ONTheFLyTESTING(S, V)

\textbf{comment:} Let IOLTS $\bar{s}$ be a specification, and let IOTS $\bar{e}$ be an environment. Let IOTS $\bar{c}$ be an implementation tested against $\bar{s}$ with respect to $\bar{e}$ by application of the following rules, initializing $S$ with $\{((\bar{s} \text{ after } \bar{e}), (\bar{e} \text{ after } \bar{e}))\}$ and verdict $V$ with None:

\begin{algorithm}
\begin{algorithmic}
  \While{$V \notin \{\text{Fail, Pass}\}$}
    \Do
      \If{$V = \text{Fail}$}
        \State{$V = \text{Pass}$}
      \EndIf
    \EndDo
  \EndWhile
\end{algorithmic}
\end{algorithm}

7.5 Closing Remarks

In this chapter, we studied the problem of deriving a specification for a third-party component, based on the specification of the system and the environment in which the component is supposed to reside. We investigated the property of \textit{decomposability} of a specification in the setting of Tretmans' \textit{ioco} theory for formal conformance testing [Tre96c]. Decomposability allows for determining whether a specification can be met by some implementation running on a given platform. Based on a new specification, to which we refer to as the \textit{quotient}, and which we derived from the given specification of the system by factoring out the effects of the platform, we identified three conditions (two on the quotient and one on the platform) that together guarantee the decomposability of the original specification.
Any component that correctly implements the quotient is guaranteed to work correctly on the given platform. However, failing implementations provide no information on the correctness of the cooperation between the component and the platform. We therefore studied strong decomposability, which further strengthens the decomposability problem to ensure that only those components that correctly implement the quotient work correctly on the given platform, meeting the overall specification. This ensures that testing a component against the quotient provides all the information needed to judge whether it will work correctly on the platform and will meet the overall specification's requirements. Thus, there is no need to retest the integrated system. However, the complexity of computing the quotient is exponential in the size of state spaces of the specification and the platform. We propose an on-the-fly test case derivation algorithm which does not compute the quotient explicitly. Components that fail such a test case provably fail to work on the platform with respect to the monolithic specification of the whole system.

Checking the inclusion relation of a platform per se is rather expensive in practice. Inspired by our studies in Chapter 5, we can show that checking the inclusion relation due to its similarity to the ioco relation has an exponential time complexity in general. As for future work, we would like to merge the two steps of checking the correctness of the platform and driving the quotient into one step and investigate whether the condition on the platform can be relaxed by ensuring that the derived quotient masks some of the unwanted behavior of the platform.

We believe that the test case generation algorithm presented in Section 4.4 can easily be generalized for $F \subseteq \text{Straces}_\mathcal{U}(s)$ for arbitrary collections $\mathcal{U}$ of non-empty output sets. However, none of our results in this chapter rely on the existence of such an algorithm; our generalization of ioco, for reasoning about relative quiescence is purely needed for developing a sound testing theory with the standard ioco relation. Moreover, in this chapter we developed the theoretical foundation for using the ioco testing framework in component-based paradigm. However, it is still required to validate the effectiveness of the developed theories in practice.
Chapter 8
Conclusions

This thesis focuses on some theoretical aspects of formal conformance testing of reactive systems. In that kind of testing, the correctness of an implementation with respect to its formal specification is often formalized using a conformance relation. Input-output conformance (ioco) is a widely-studied and commonly-used conformance relation. In Chapter 3, an industrial experience in applying the ioco testing framework on an EFT switch was reported to illustrate some limitations of the ioco relation in testing concurrent systems. The obtained results from that experience demonstrate the effectiveness of the ioco testing framework as well as the need for several improvements, which were addressed in Chapters 4, 5, 6, and 7.

In Chapter 4, the ioco relation was compared with the implementation relation introduced in [WW09; Wei09]. The latter relation was studied in the literature to cater for asynchronous input-output conformance testing. Using the ioco relation when testers and the IUT communicate asynchronously, testing may result in a false fail verdict, i.e., testing may report a failure while the behavior of the implementation conforms with the specification, see Section 3.4. It was investigated when test results of those implementation relations coincide, so that they can be used interchangeably in testing. To carry out the comparison, it was essential to formally define the testing power of each relation. In [WW09; Wei09], the notion of conformance was only extensionally defined, based on the execution of internal-choice test cases. As one of the contributions of Chapter 4, therefore, the respective intensional definition of the notion of conformance was given. The results of that chapter showed that those implementation relations coincide only in a restricted setting: when the behavioral model of the IUT and the specification are limited to a subclass of IOLTSs, namely internal-choice IOLTSs. Therefore, using internal choice test cases, does not necessarily lead to sound verdicts with respect to the ioco relation even in the synchronous setting.

To have a better insight into the ioco relation, this relation was studied for the first time from a computational point of view in Chapter 5. It was shown that the problem of deciding the ioco relation between IOLTSs in general is PSPACE-complete. Next, a polynomial time algorithm for checking the ioco relation in a restricted setting, namely deterministic specifications, was presented. To obtain this result, the ioco relation was
redefined as a coinductive (simulation-like) relation. The presented polynomial time algorithm in this section is based on the polynomial-time reduction of the problem of deciding the coinductive ioco relation, respectively, to and from the NHORNSAT problem.

Chapter 6 thoroughly investigated the problem of asynchronous communication between testers and the IUT in the ioco framework. It is well known in the literature that standard ioco test cases may result in a false fail verdict in the asynchronous setting, see Section 3.4. The first solution in the literature to this problem is to compose a given specification with two FIFO queues as models of asynchronous channels. An obvious drawback of this solution is the infamous state space explosion problem. To sidestep this problem, either the class of test cases, the class of specification, the class of implementations, or a combination thereof has to be restricted.

To avoid this problem, first a restriction on test cases, namely internal-choice test cases, is studied. It was shown in Chapter 6 that for internal-choice IOLTS specifications and implementations behaving as internal-choice IOTs this approach leads to a sound and complete theory under very natural conditions, thereby a rigorous proof was given for the earlier results of [WW09; Wei09]. However, it was argued by a counterexample that the results in [Wei09], using internal-choice test cases, cannot be applied to the ioco testing framework. Afterwards, two subclasses of IOLTSs were identified, respectively, for implementations and specifications whose test results obtained from standard ioco test cases in the asynchronous setting are the same as those obtained in the asynchronous setting.

Chapter 7 studied the problem of extracting the specification of an unknown subsystem/component of a compound system from a given monolithic specification of the system with respect to the known components, collectively called platform. In this regard, two characteristics of decomposability and strong decomposability were identified for a given a specification \( \bar{s} \) with respect to a given platform \( \bar{\epsilon} \). Decomposability determines whether a specification \( \bar{s} \) can be met by a component \( c \) integrating to platform \( \bar{\epsilon} \), and strong decomposability determines whether there is a specification that distinguishes all possible component \( c \) whose composition with platform \( \bar{\epsilon} \) conforms with specification \( \bar{s} \). A constructive approach to derive the specification of the unknown component from specification \( \bar{s} \) and platform \( \bar{\epsilon} \) was presented in this chapter. Afterwards, the necessary and sufficient conditions were identified on the extracted specification to guarantee decomposability and strong decomposability. Moreover, an on-the-fly algorithm for testing the unknown component was presented based on strong decomposability to avoid constructing the specification of the unknown component explicitly upfront.

**Future work** The theories developed in this thesis serve as theoretical foundations for solutions to improve the ioco testing framework in testing concurrent systems. It is recommended as future work to evaluate the developed theories in this thesis in practice.

The results in Chapter 6 serve as a basis for developing practical solutions for the asynchrony problem in the ioco testing framework. To benefit from the results developed in Section 6.3 in practice, an algorithm for checking the robustness property on IOLTSs has to be developed and implemented. The results in Chapter 5 can be used for this purpose.
There are two main directions in future research which are aimed at extending the result of checking the robustness property of a specification. The first direction is to find the maximal sub-specification of a non-robust specification fulfilling the robustness conditions. This sub-specification represents a subset of the behavior of the specification, which can be used in testing in the asynchronous setting. The second direction is to complete a non-robust specification such that it becomes robust. For example, the application of the queue operator on a specification always results in a robust specification. However, the queue operator is not an efficient solution due to its drawbacks mentioned in this thesis. Hence, finding an operator that computes the minimal robust completed specification, which does not have the state-space explosion problem is relevant future work.

Using the theories developed in Chapter 7, an on-the-fly algorithm for testing an unknown component of a system was presented. To evaluate the effectiveness of solutions presented in Chapter 7, this algorithm needs to be implemented and used in some practical case studies. As mentioned in Chapter 7, the current algorithm has an exponential time complexity. Even if the exponential complexity is not an obstacle in using this algorithm in practice, improving the complexity of the results of Chapter 7 is relevant future work.

Moreover, to enhance the use of model-based testing in industry, further research on theoretical aspects of model-based testing theories is required. In the rest of this chapter, some ideas for future work are discussed.

One of the topics that can substantially help to improve model-based testing practices in industry is the test selection problem: selection of a finite subset of the exhaustive set of test cases such that the testing power of the selected subset is the same as that of the exhaustive set. The exhaustive set of test cases generated by the ioco test case generation algorithm is typically infinite. Since checking an infinite set of test cases is impossible, the test selection problem has gained considerable research interest among the testing community [GG75; WO80; Gau95; SP14; RC85]. Unlike white-box testing, test cases in black-box testing are derived from a specification of a system without any information about the internal structure of the system. As stated in [Gau95], the test selection problem thus needs more hypotheses on implementations, which are referred to as test selection hypotheses. Two types of test selection hypotheses are identified in [Gau95]: the regularity hypothesis and the uniformity hypothesis.

The regularity hypothesis formalizes the assumption that checking test cases of at most a certain level of complexity is enough to assess the correctness of an implementation. Translating the regularity hypothesis in the context of the ioco relation leads to identifying a limit on the length of test cases: examining test cases under a certain length is sufficient to assess the conformance of an implementation with a specification. As argued in the literature [Hie+09; Gau95], imposing any condition on test cases derived just from a specification of the system is only possible by making stronger assumptions about the implementation. In this sense, it would be worthwhile to investigate how to find a set of practical assumptions that can be assumed about the structure (behavior) of the implementation under test such that the regularity hypothesis can be established for the ioco test cases. Our result in Chapter 5 on the complexity of checking the ioco relation may serve a basis for addressing this problem.

In most domains, data are an intrinsic part of systems. In recent years therefore,
many studies have focused on extending testing theories with data [FTW06; FTW04; FGL07; Fai+08; LRT07; Ban+12]. In this regard, the \texttt{sioco} relation [FTW06] has been developed as the extension of the \texttt{ioco} relation with data. However, testing theories and tools developed so far are still immature in the data selection aspect. The uniformity hypothesis is the formal counterpart of some common black-box testing methods like data partitioning equivalence. It states that in testing there is no need to check all instances of a data variable but only the representatives of each and every sub-domain. Since data selection is a major challenge in applying formal conformance testing in practice, future work is required on data selection criteria in line with the uniformity hypothesis. To our knowledge, Huang and Peleska in [HP13] are the first to present a formal approach to combine data equivalence partition testing with formal conformance testing of reactive systems. However, the notion of conformance used in [HP13] is different from the \texttt{sioco} relation. Thus, it is recommended to investigate whether a similar approach to [HP13] can be applied to the data selection problem in the framework the \texttt{sioco} testing framework.
Appendix A

Proofs of Formal Results in Chapter 6

Lemma 6.6. Let \( \langle S, I, U, \rightarrow_s, \hat{s} \rangle \) be an IOTS\(^\dagger\), and let \( \langle T, U \cup \{\theta\}, I, \rightarrow_t, \hat{t} \rangle \) be a TTS\(^\dagger\). Let \( s, s' \in S \), and \( t, t' \in T \) be arbitrary states, and let \( \sigma_u \in U^* \) and \( \sigma_i \in I^* \) and \( a \in I \). If \( t \|_{[\sigma_u \ll s \ll \sigma_i]} \overset{a}{\Rightarrow} t' \|_{[\sigma_u \ll s' \ll \sigma_i]} \), then \( \delta_q([\sigma_u \ll s' \ll \sigma_i]) \).

Proof. Assume \( a \in I \) and \( t \|_{[\sigma_u \ll s \ll \sigma_i]} \overset{a}{\Rightarrow} t' \|_{[\sigma_u \ll s' \ll \sigma_i]} \), we know there exists an \( s'' \in S \) such that \( t \|_{[\sigma_u \ll s'' \ll \sigma_i]} \overset{a}{\Rightarrow} t' \|_{[\sigma_u \ll s'' \ll \sigma_i]} \). It follows from Definition 2.31 together with Definition 2.11 that \( s \overset{a}{\Rightarrow} s'' \) and also \( s'' \overset{a}{\Rightarrow} s' \). We thus find that \( s \overset{a}{\Rightarrow} s' \) and subsequently according deduction rule I1 in Definition 2.11, we have \( [\sigma_u \ll s \ll \sigma_i] \overset{a}{\Rightarrow} [\sigma_u \ll s' \ll \sigma_i] \). The former observation leads to \( t \|_{[\sigma_u \ll s \ll \sigma_i]} \overset{a}{\Rightarrow} t' \|_{[\sigma_u \ll s' \ll \sigma_i]} \). Using deduction rule A1 in Definition 2.11 and applying deduction rule R2' in Definition 2.31 result in \( t \|_{[\sigma_u \ll s' \ll \sigma_i]} \overset{a}{\Rightarrow} t' \|_{[\sigma_u \ll s' \ll \sigma_i]} \). Hence, there is a trace starting from \( t \|_{[\sigma_u \ll s \ll \sigma_i]} \) to \( t' \|_{[\sigma_u \ll s' \ll \sigma_i]} \). It follows then from Definition 2.33 that \( \delta_q([\sigma_u \ll s' \ll \sigma_i]) \) (since test case \( t \) only provides an input immediately after if it has observed quiescence), which was to be shown.

Lemma 6.7. Let \( \langle S, I, U, \rightarrow_s, \hat{s} \rangle \) be an IOTS\(^\dagger\), and let \( \langle T, U \cup \{\theta\}, I, \rightarrow_t, \hat{t} \rangle \) be a TTS\(^\dagger\). Let language \( L' = U \cup I \cup \{\theta\} \) and let \( s, s' \in S \), \( t, t' \in T \) be arbitrary states. There is no trace \( \sigma_u \in L'^* \) such that \( t \|_{\sigma_u \ll s \ll \sigma_i} \overset{\sigma}{\Rightarrow} t' \|_{\sigma_u \ll s' \ll \sigma_i} \) and the input and output queues are both non-empty at the same time \( (\sigma_i \neq \varepsilon \land \sigma_u \neq \varepsilon) \).

Proof. Assume, towards a contradiction, that the following two statements hold:

1. \( t \|_{\sigma_u \ll s' \ll \sigma_i} \)
2. \( \sigma_i \neq \varepsilon \land \sigma_u \neq \varepsilon \)

Since both \( \sigma_i \) and \( \sigma_u \) are non-empty, there must exist the largest prefix \( \sigma' \) of \( \sigma \) during which the two queues are never simultaneously non-empty, i.e., by observing a single
action after $\sigma'$, both queues become non-empty for the first time. Hence, there exists $\sigma', \sigma'' \in L^*$ as a prefix and postfix of $\sigma$ respectively and $y \in L' \cup \{\tau\}$ such that $\sigma = \sigma'y\sigma''$ and we have,

1. $\tau \| Q(s) \xrightarrow{\sigma'} t_1 \| [\sigma'_t, s_1 \ll \sigma'_t]$ with $t_1 \in T$ and $s_1 \in S$ for some $\sigma'_t \in I^*$ and $\sigma''_u \in U^*$ such that only one of $\sigma'_t$ and $\sigma''_u$ are non-empty, i.e., $(\sigma'_t = \epsilon \land \sigma'_t \neq \epsilon)$ or $(\sigma''_u = \epsilon \land \sigma''_u \neq \epsilon)$

2. $t_1 \| [\sigma'_t, s_1 \ll \sigma'_t] \xrightarrow{\gamma} t_2 \| [\sigma''_u, s_2 \ll \sigma''_u]$ with $t_2 \in T$ and $s_2 \in S$ for some $\sigma''_u \in I^*$ and $\sigma''_u \in U^*$ such that the empty queue gets non-empty at the end of this transition i.e., $(\sigma''_u = \epsilon \implies \sigma''_u \neq \epsilon)$, or similarly $(\sigma''_u = \epsilon \implies \sigma''_u \neq \epsilon)$

3. $t_2 \| [\sigma''_u, s_2 \ll \sigma''_u] \xrightarrow{\sigma''} t' \| [\sigma''_u, s' \ll \sigma''_u]$.

Note that after $\sigma'$ both input and output queues cannot be empty, since a single transition $\gamma$ only increases the size of one of the two queues (see rules $A1$ and $I2$ in Definition 2.11). Below, we distinguish two cases based on the status of the input queue after executing the trace $\sigma'$: either the input queue is empty (and the output queue is not), or the other way around.

1. Case $\sigma'_t = \epsilon$. The only possible transition that can fill an output queue is due to the application of deduction rule $I2$ in Definition 2.11. Hence, there must exist some $s_2$ and $x \in U$ such that $[\epsilon \ll s_1 \ll \sigma'_t] \xrightarrow{\tau} [x \ll s_2 \ll \sigma'_t]$ and subsequently, $(t_1 \| [\epsilon \ll s_1 \ll \sigma'_t] \xrightarrow{\tau} t_2 \| [x \ll s_2 \ll \sigma'_t])$ (thereby satisfying the third item with $\sigma''_u = \epsilon$ and $\sigma''_u = x$). The former $x$-labeled transition can only be due to deduction rule $I2$ in Definition 2.11 and hence, we have $s_1 \xrightarrow{x} s_2$. However, it follows from $\sigma'_t \neq \epsilon$ that there exists an $a \in I$, $s_p \in S$, a prefix $\sigma'_p$ of $\sigma'$ and $\rho_i \in I^*$ such that $\sigma'_i = \rho_i a$ and $t \| Q(s) \xrightarrow{\sigma'_i} t' \| [\epsilon \ll s_p \ll \rho_i].$ We have from Lemma 6.6 that $\delta_q([\epsilon \ll s_1 \ll \rho_i]).$ Using deduction rule $A2$ on $s_1 \xrightarrow{x} s_2$, we obtain that $[\epsilon \ll s_1 \ll \rho_i] \xrightarrow{\epsilon} [x \ll s_2 \ll \rho_i].$ Hence according to Definition 2.12, state $[\epsilon \ll s_1 \ll \rho_i]$ is not quiescent, which contradicts our observation that $\delta_q([\epsilon \ll s_1 \ll \rho_i]).$

2. Case $\sigma'_t = \epsilon$. The only transition which allows for filling the input queue is due to the subsequent application of deduction rules $R2$ and $A1$ in Definitions 2.29 and 2.21 respectively. Hence, there exists an $a \in I$, such that $t_1 \| [\sigma'_t, s_1 \ll \epsilon] \xrightarrow{a} t_2 \| [\sigma'_t, s_2 \ll a]$ and $[\sigma'_t, s_1 \ll \epsilon] \xrightarrow{a} [x \ll s_2 \ll \epsilon]$ (where the former satisfies the third item by taking $\sigma'_t = \epsilon$ and $\sigma''_u = a$). It follows from Lemma 6.6 that $\delta_q([\sigma'_t, s_1 \ll \epsilon]).$ However since $\sigma'_t \neq \epsilon$, there exists a $x \in U$ and $\rho_u \in U^*$, such that $\sigma''_u = x \rho_u$ and using deduction rule $A2$ in Definition 2.11, we obtain that that $[\sigma'_t, s_2 \ll \epsilon] \xrightarrow{x} [\sigma''_u, s_2 \ll \epsilon]$, and thus, $[\sigma''_u, s_2 \ll \epsilon]$ is not quiescent, which contradicts our earlier observation.

\[\square\]

**Lemma 6.16.** Let $(S, I, U, \rightarrow, s)$ be an IOTS over language $L = I \cup U$, and let $s \in S$ and $\sigma \in L_\sigma^*$. Then $\sigma \in \mathcal{Staces}(Q(s))$ implies that there is a $s' \in S$ such that $Q(s) \xrightarrow{\sigma} Q(s')$.
Proof. The proof is given by induction on the number of δ in σ ∈ L_δ^∗.

- Base case. We assume the number of δ in σ is 0, i.e., σ ∈ L^∗. We distinguish between two cases based on whether σ ∈ I^* and σ ∈ I^*. 

1. Case σ ∈ I^* : Due to deduction rule A1 in Definition 2.11, it always holds that Q(s) ⇒ [ε ≪ s ≪ σ]. Since s is input enabled, there is a state s’ ∈ S such that s ⇒ s’. By applying deduction rule I3 several times, we have [ε ≪ s ≪ σ] ⇒ Q(s’). We thus find that s’ meets the required condition.

2. Case σ /∈ I^* : Let σ = σ’xρ, with σ’ ∈ L^*, x ∈ U and ρ ∈ I^*. The appearance of x in trace σ’xρ can only be due to deduction rules I2 and A2 in Definition 2.11 and hence, we should have 

\[ Q(s) \xrightarrow{\sigma_1} [\sigma_2] \xrightarrow{\sigma_2} [\sigma_3] \xrightarrow{\sigma_3} \]  

for σ_w, σ_u, σ_v, U_v, σ_k, σ_i, σ_j ∈ I^* and s”, s_1, s_2, s_3 ∈ S. We conclude from the last observation and deduction rules A2 in Definition 2.11 that σ_2 must be the projection of σ_2 onto U^*. It follows from the last observation and deduction rules A1 and A2 that also the following derivation is possible, [ε ≪ s_2 ≪ σ_2] ⇒ [ε ≪ s_3 ≪ σ_2ρ], where σ_2’ is the projection of σ_2 onto I^*. Since, s_2 is input-enabled there is a state s’ ∈ S such that s_2 ⇒ s’. By using deduction rule I3, we have [ε ≪ s_2 ≪ σ_2’ρ] ⇒ Q(s’). Thus s’ meets the required condition.

- Induction step. We assume that the statement holds for all σ’ ∈ L_δ^* with at most n - 1 occurrences of δ. Suppose the number of occurrences of δ in σ is n. Since σ ∈ Straces(Q(s)), there exists a state s” ∈ S such that Q(s) ⇒ [σ_1] and for some σ_1 ∈ I^* and σ_2 ∈ L^*. Assume σ = σ_1δσ with σ_1 ∈ L^* and σ_2 ∈ L_δ^*. Due to definition of suspension traces, the following step has to be taken in the former derivation, 

\[ Q(s) \xrightarrow{\sigma_1} [\sigma_2] \xrightarrow{\sigma_2} [\sigma_3] \xrightarrow{\sigma_3} \]  

where δ_2 ∈ [σ_1 ≪ s_1 ≪ σ_1] for some s_1 ∈ S, σ_2 ∈ U^* and σ_2 ∈ I^*. Note that σ_2 has to be empty since quiescence has been observed beforehand. It follows from Definition 2.12 that σ_2 has to be empty as well, since otherwise, [σ_1 ≪ s_1 ≪ σ_1] can perform an internal transition, hence it cannot be quiescent. We thus find that Q(s) ⇒ Q(s_1) and s_1 is quiescent. We take the last transition of the previous derivation. It follows from the induction hypothesis that ∃s’ ∈ S such that Q(s_1) ⇒ Q(s’). We thus conclude from the last observation that there is a state s’ ∈ S such that Q(s) ⇒ Q(s_1) ⇒ Q(s’) which was to be shown. 

\[ \square \]

The above lemma ensures that any asynchronous trace execution can lead to a state where both input and output queues are empty. Subsequently, the following lemma shows that the same state is also reachable in the synchronous setting just by executing a trace that reorders actions of the asynchronous trace; shifts outputs before inputs.
Lemma 6.17. Let \(\langle S, I, U, \rightarrow, \bar{s} \rangle\) be an IOTS. Let \(s, s' \in S\) and \(\sigma \in \text{Straces}(Q(s))\). Then \(Q(s) \overset{\sigma}{\Rightarrow} Q(s')\) implies there is a \(\sigma' \in \text{Straces}(s)\) such that \(s \overset{\sigma'}{\Rightarrow} s'\) and \(\sigma' @ \sigma\).

Proof. The proof is given by induction on the number of \(\delta\) in \(\sigma \in L_\delta^\ast\).

- Base case. We assume that there is no occurrence of \(\delta\), i.e., \(\sigma \in L^\ast\). Thus, the thesis reduces to \(\sigma \in \text{Straces}(Q(s))\) and \(\sigma \in L^\ast\) implies there is a \(\sigma' \in \text{traces}(s)\) such that \(\sigma' @ \sigma\). We prove the latter by induction on the number of output actions in \(\sigma \in L^\ast\).

- Inductive step. We assume that the statement holds for all \(\sigma'' \in L^\ast\) with at most \(n - 1\) output actions. Suppose that the number of output actions of \(\sigma\) is \(n\).

Assume that \(\sigma = \rho x \bar{\sigma}\) with \(\rho \in L^\ast\), \(x \in U\) and \(\bar{\sigma} \in L^\ast\). We have \(Q(s) \overset{\rho x \bar{\sigma}}{\Rightarrow} Q(s')\), implying that somewhere in this derivation the step \(s_1 \overset{x}{\Rightarrow} s_2\) is taken, for some \(s_1, s_2 \in S\). This implies that there are two prefixes \(\rho_1\) and \(\rho_2\) of \(\rho\) such that \(\rho_2\) is a prefix of \(\rho_1\) as well and also \(Q(s) \overset{\rho_1}{\Rightarrow} Q(s')\). The last step of the previous derivation and deduction rule \(A2\) in Definition 2.11 lead to \(Q(s')\). Since the input queue can be filled only under deduction rule \(A1\) in Definition 2.11 that \(Q(s_2) \overset{\rho_2}{\Rightarrow} Q(s')\). By defining \(\sigma_1 = \rho \setminus \rho_2\bar{\sigma}\), we have \(Q(s_2) \overset{\sigma_1}{\Rightarrow} Q(s')\) with \(\sigma_1 \in L^\ast\) and one output action less than \(n\). It follows from induction hypothesis that \(\exists \sigma'_1 \in \text{Straces}(s_2)\bullet^\ast \sigma'_1 \Rightarrow s' \wedge \sigma'_1 @ \sigma_1\). We thus have \(s \overset{\rho_2}{\Rightarrow} s_1 \overset{x}{\Rightarrow} s_2 \overset{\sigma'_1}{\Rightarrow} s'\) and subsequently, \(\rho_2 x \sigma'_1 \in \text{Straces}(s)\) by applying deduction rule \(REF\) and \(COM\) in Definition 6.14 respectively, we have \(x \sigma'_1 @ x (\rho \setminus \rho_2)\sigma_1\). On the other hand, due to rule \(REF\) and \(COM\) we know that \(x (\rho \setminus \rho_2)\sigma_1 @ (\rho \setminus \rho_2)x \sigma_1\) and consequently, \(x \sigma'_1 @ (\rho \setminus \rho_2)x \sigma_1\). Deduction rule \(COM\), the last observation and \(\rho_2 @ \rho_2\) lead to \(\rho_2 x \sigma'_1 @ \rho_2 (\rho \setminus \rho_2)\sigma_1\). By defining \(\sigma' = \rho_2 x \sigma'_1\), we have \(\sigma' @ \rho_2 (\rho \setminus \rho_2)\sigma_1\) and more clearly, \(\sigma' @ \sigma\). We thus find that \(\sigma'\) meets the two desired conditions.

- Inductive step. We assume the statement holds for all \(\sigma\) with at most \(n - 1\) occurrences of \(\delta\). Suppose there are \(n\) occurrences of \(\delta\) in \(\sigma\). Assume \(\sigma = \sigma_1 \delta \bar{\sigma}\) with \(\sigma_1 \in L^\ast\) and \(\bar{\sigma} \in L_\delta^\ast\). By Proposition 6.16, we know from \(\sigma \in \text{Straces}(s)\) that there is a state \(s' \in S\) such that \(Q(s) \overset{\sigma_1 \delta}{\Rightarrow} Q(s')\). Due to Definition 2.12 and Definition 2.14, there exists a state \(s_1 \in S\) such that \(Q(s) \overset{\sigma_1 s_1}{\Rightarrow} Q(s_1) \overset{\sigma}{\Rightarrow} Q(s')\) and \(\delta(s_1)\). By taking the first transition of the previous derivation and induction basis, we find that there exists \(\sigma'_1 \in \text{Straces}(s)\) such that \(s \overset{\sigma'_1}{\Rightarrow} s_1\) and \(\sigma'_1 @ \sigma\). From \(\delta(s_1)\), we have \(s \overset{\sigma'_1}{\Rightarrow} \delta(s_1)\).
and consequently by applying deduction rule $COM$ in Definition 6.14, $\sigma'_1 \delta @ \sigma_1 \delta$ is concluded. Take then, the last transition of the first derivation i.e, $Q(s_1) \xRightarrow{\bar{\delta}} Q(s')$ with $\bar{\sigma} \in L''_\delta$ and the number of occurrences of $\delta$ is $n - 1$ (one less than $\sigma$). By our induction hypothesis we find that there exists a $\bar{\sigma}' \in \text{Straces}(s_1)$ such that $s_1 \xRightarrow{\bar{\sigma}'} s'$ and $\bar{\sigma}' @ \bar{\sigma}$. We thus have $\exists \sigma'_1 \in \text{Straces}(s), \bar{\sigma}' \in \text{Straces}(s_1) \bullet_{\delta} \xRightarrow{\bar{\sigma}'} s_1 \xRightarrow{\bar{\sigma}'} s'$. By applying deduction rule $COM$ to the first and second observation, i.e., $\sigma'_1 \delta @ \sigma_1 \delta$ and $\bar{\sigma}' @ \bar{\sigma}$, we have $\sigma'_1 \delta \bar{\sigma}' @ \sigma_1 \delta \bar{\sigma}$. By defining $\sigma' = \sigma'_1 \delta \bar{\sigma}'$ we find that $\sigma'$ satisfies the two required properties.

$\blacksquare$
Appendix B

Proofs of Formal Results in Chapter 7

Proposition 7.8. Let $\langle S, I, U, \rightarrow_s, \bar{s} \rangle$ be an IOLTS. Let $V \subseteq U$ and $\Sigma \subseteq \mathcal{P}(U)$ such that $V$ is a subset of all member of $\Sigma$, i.e., $\forall O \in \Sigma \cdot V \subseteq O$. We then have for all $s \in S$ and all $\sigma \in (L^*_U \setminus V)^*$:

$$(\text{hide}[V][\bar{s}]) \text{ after } \Sigma \sigma = \{ (\text{hide}[V][s']) \mid s' \in \bar{s} \text{ after } \Sigma \sigma' \land \sigma' \in L^*_U \land \sigma = \sigma'_{L^*_U \setminus V} \}$$

Proof. We divide the proof obligation into two parts:

1. $\forall \sigma \in (L^*_U)^* \cdot (\text{hide}[V][\bar{s}]) \text{ after } \Sigma \subseteq \{ s' \in \bar{s} \text{ after } \Sigma \sigma' \land \sigma' \in L^*_U \land \sigma = \sigma'_{L^*_U \setminus V} \}$
2. $\forall \sigma \in (L^*_U)^* \cdot \{ s' \in \bar{s} \text{ after } \Sigma \sigma' \land \sigma' \in L^*_U \land \sigma = \sigma'_{L^*_U \setminus V} \} \subseteq (\text{hide}[V][\bar{s}]) \text{ after } \Sigma$

We prove each of them in turn.

1. We prove that $\forall \sigma \in (L^*_U)^* \cdot (\text{hide}[V][\bar{s}]) \text{ after } \Sigma \subseteq \{ s' \in \bar{s} \text{ after } \Sigma \sigma' \land \sigma' \in L^*_U \land \sigma = \sigma'_{L^*_U \setminus V} \}$.

We show that $\forall \sigma \in (L^*_U)^* \cdot \forall q \in (\text{hide}[V][\bar{s}]) \text{ after } \Sigma \cdot q \in \{ s' \in \bar{s} \text{ after } \Sigma \sigma' \land \sigma' \in L^*_U \land \sigma = \sigma'_{L^*_U \setminus V} \}$.

The proof proceeds by induction on the length of $\sigma$.

- Base case. For the base case, we assume that $\sigma = \epsilon$, i.e., $\text{hide}[V][\bar{s}] \Rightarrow q$. We use the second induction on the number of $\tau$-transition leading to $\epsilon$. For the base case of the second induction, we suppose that $q = \text{hide}[V][\bar{s}]$; thus $q = \bar{s}$. We take $\sigma' = \epsilon$; thus $\sigma'_{L^*_U \setminus V} = \epsilon$. Therefore, $q \in s$ after $\sigma'$; thus $q \in \{ s' \in \bar{s} \text{ after } \sigma' \land \sigma' \in L^*_U \land \sigma = \sigma'_{L^*_U \setminus V} \}$.

We assume for the induction step of the second induction that the above thesis holds for all sequence of $\tau$-transition with length of $n-1$ and the number of $\tau$-transition leading to $\Rightarrow$ is $n$. Therefore, there exists $s''$ such that $\text{hide}[V][\bar{s}] \Rightarrow s'' \Rightarrow q$, where $\sigma_{\bar{s}}$ including $n-1 \tau$-transition. Following
the induction hypothesis, we obtained that there exists $\sigma'' \in L^*_u$ such that $s'' \in \tilde{s}$ after $\sigma''$ and $\sigma''_{I_{u|V}} = \sigma$. Regarding Definition 7.7, $s'' \xrightarrow{\rho} q$ in IOLTS $\text{hide}[V] \text{in} \tilde{s}$ results in $s'' \xrightarrow{\rho} q$ in IOLTS $\tilde{s}$ or there exists $x \in V$ such that $s'' \xrightarrow{\rho} q$. We first suppose that $s'' \xrightarrow{\rho} q$ in IOLTS $\tilde{s}$; thus $q \in \tilde{s}$ after $\sigma''$ with $\sigma''_{I_{u|V}} = \sigma$. Therefore, $q \in \{s' \in \tilde{s} \text{ after } \sigma' \mid \sigma' \in L^*_u \land \sigma'_{I_{u|V}} = \sigma\}$.

Next, we suppose that $s'' \xrightarrow{\rho} q$ with $v \in V$. It follows from Definition 7.4 that $(\sigma'')_{I_{u|V}} = \sigma \land \sigma'_{I_{u|V}} = \sigma$. Therefore, $q \in \{s' \in \tilde{s} \text{ after } \sigma' \mid \sigma' \in L^*_u \land \sigma'_{I_{u|V}} = \sigma\}$.

**Induction step.** We assume for the induction step that the above thesis holds for all sequences with the length of $n-1$ and that the length of $\sigma$ is $n$. We take $\sigma = \rho x$ with $\rho \in (L_u \setminus V)^*$ and $x \in (L_u \setminus V)$. Therefore, there exists $q'$ such that $\text{hide}[V] \text{in} \tilde{s} \xrightarrow{\rho} q'' \xrightarrow{\rho} q$.

Following induction hypothesis there exists $\rho' \in L^*_u$ such that $\tilde{s} \xrightarrow{\rho'} q''$ and $\rho'_{I_{u|V}} = \rho$. We conclude from $q'' \xrightarrow{\rho} q$ that there exist $q'_1$ and $q''_2$ such that $q'' \xrightarrow{\rho} q'_1 \xrightarrow{\rho} q''_2$ and $q'_1 \xrightarrow{x} q''_2$ with $x \in L_{u|V}$. Therefore, it is obtained that $\tilde{s} \xrightarrow{\rho'\rho''x} q''_2$ whereas $(\rho'\rho''x)_{I_{u|V}} = \rho x$. We consider two cases based on $x \in \delta_u$ or $x \notin \delta_u$. We first assume that $x \in \delta_u$ which results in $q''_1$ is stable; thus, $q''_1 = q''_2$. We second assume that $x \notin \delta_u$. In the similar lines of reasoning in the base case, it is proven that $\exists \rho'' \in L^*_u$ such that $q''_2 \xrightarrow{\rho''} q$ and $\rho''_{I_{u|V}} = \epsilon$. Therefore, $\tilde{s} \xrightarrow{\rho'\rho''x} q$ whereas $\rho'\rho''x_{I_{u|V}} = \rho x$ is clearly deduced. Hence, $q \in \{s' \in \tilde{s} \text{ after } \sigma' \mid \sigma' \in L^*_u \land \sigma'_{I_{u|V}} = \sigma\}$.

2. In order to prove $\forall \sigma \in (L_u)^* \bullet \{s' \in \tilde{s} \text{ after } \sigma' \mid \sigma' \in L^*_u \land \sigma = \sigma'_{I_{u|V}}\} \subseteq (\text{hide}[V] \text{in} \tilde{s})$ after $\sigma$, we show that $\forall \sigma \in (L_u)^* \bullet \forall q \in \{s' \in \tilde{s} \text{ after } \sigma' \mid \sigma' \in L^*_u \land \sigma = \sigma'_{I_{u|V}}\} \bullet q \in (\text{hide}[V] \text{in} \tilde{s})$ after $\sigma$. The proof proceeds by induction on the length of $\sigma$.

**Base case.** We assume for the base case that $\sigma = \epsilon$. Therefore, there exists $\sigma' \in L^*_u$ such that $s \xrightarrow{\sigma'} q$ and $\sigma'_{I_{u|V}} = \epsilon$; thus $\sigma' \in V^*$. The proof continues with the second induction on the length of $\sigma'$. We suppose for the base case of the second induction that $\sigma = \epsilon$; thus $s \xrightarrow{\epsilon} q$. Regarding Definition 7.7, $(\text{hide}[V] \text{in} \tilde{s}) \xrightarrow{\epsilon} q$ is resulted. Hence, $q \in (\text{hide}[V] \text{in} \tilde{s})$ after $\sigma$ with $\sigma = \epsilon$.

We assume for the induction step of the second induction that the above thesis holds for all sequences with the length of $n - 1$ and the length of $\sigma'$ is $n$. We take $\sigma' = \rho x$ with $\rho' \in V^*$ and $x \in V$. Therefore, there exists $q', q''_1$ and $q''_2$ such that $s \xrightarrow{\rho'} q' \xrightarrow{\epsilon} q''_1 \xrightarrow{x} q''_2 \xrightarrow{\epsilon} q$. Following induction hypothesis, it is
obtained that \( q_i'' \in (\text{hide}[V] \text{in}s) \) after \( \sigma \). It is concluded from Definition 7.7 that \( q_i' \xrightarrow{\delta} q'' \), regarding \( x \in V \). Consequently, it is obtained from \( q_2'' \xrightarrow{\varepsilon} q \) that \( q_i' \xrightarrow{\delta} q \). Therefore, \( q \in (\text{hide}[V] \text{in}s) \) after \( \sigma \) is clearly resulted.

- **Induction step.** We assume for the induction step of the second induction that the above thesis holds for sequences with the length of \( n - 1 \) and the length of \( \sigma \) is \( n \). We take \( \sigma = \rho x \) with \( \rho \in (L_{\equiv})^* \) and \( x \in (L_{\equiv}) \). Therefore, there exists \( \sigma' \in (L_{\equiv})^* \) such that \( s \xrightarrow{\sigma'} q \) and \( \sigma' \downarrow_{L_{\equiv}} = \sigma \). We chop \( \sigma' \) to \( \rho' \) and \( \rho_x \) regarding \( \rho \) and \( x \) such that \( \rho \downarrow_{L_{\equiv}} = \rho \) and \( \rho \downarrow_{L_{\equiv}} \) is the largest prefix of \( \rho' \) such that \( \rho' \downarrow_{L_{\equiv}} = \rho \); thus \( \sigma_x = x\sigma_v \) with \( \sigma_v \in V^* \). Therefore, there exist \( q', q'' \) such that \( \bar{s} \xrightarrow{\rho'} q' \xrightarrow{x} q'' \xrightarrow{\sigma} q \). Following induction hypothesis, it is obtained that \( q' \in (\text{hide}[V] \text{in}s) \) after \( \rho \). It is concluded from \( q' \xrightarrow{x} q'' \) with \( x \notin V \) that \( q'' \in (\text{hide}[V] \text{in}s) \) after \( \rho x \). We consider that \( x \in \delta_{\equiv} \) is possible in \( (\text{hide}[V] \text{in}s) \) due to \( \forall Y \in \equiv \cdot V \subseteq Y \). We consider two cases; \( x \in \delta_{\equiv} \) and \( x \notin \delta_{\equiv} \). We first assume that \( x \in \delta_{\equiv} \); thus, \( q'' = q \) because \( q'' \) is stable. We second assume that \( x \notin \delta_{\equiv} \). In similar lines of reasoning with respect to \( \sigma \downarrow_{L_{\equiv}} = \varepsilon \), it is proven that \( q'' \xrightarrow{\varepsilon} q \). Therefore, \( q \in (\text{hide}[V] \text{in}s) \) after \( \rho x \) is clearly obtained.

\[ \square \]

We recall Theorem 7.14 from the body of the Paper below.

**Theorem 7.14.** Let \( \langle S, I \cup U \cup \delta_{\equiv}, \rightarrow, \bar{s} \rangle \) be a valid extended suspension automaton. Then, there is an IOLTS \( \langle S', I, U, \rightarrow, \bar{s}' \rangle \) over language \( L = I \cup U \), such that

- \( \text{traces}(\bar{s}) = \text{Straces}_{\equiv}(\bar{s}') \).
- for all \( F \subseteq \text{traces}(\bar{s}) \) and all implementations \( \bar{r} \), \( \bar{r} \text{ioco}_F \bar{s} \iff \bar{r} \text{ioco}_F \bar{s}' \).

Before we address the proof of the above theorem we need to show the correctness of the lemma given below.

**Lemma B.1.** Let \( \langle S, I \cup U \cup \delta_{\equiv}, \rightarrow, \bar{s} \rangle \) be a valid extended suspension automaton. We define the relation \( \equiv \) on state space \( S \) as \( s \equiv t \) iff \( \text{traces}(s) \subseteq \text{traces}(t) \) and we define the equivalence relation \( \equiv \) on states as \( s \equiv t \) iff \( s \equiv t \) and \( t \equiv s \). We show the \( \equiv \)-equivalence class of \( s \) by \( [s]_{\equiv} \). Let \( M = \langle S_{\equiv}, I, U, \rightarrow_{L}, [\bar{s}]_{\equiv} \rangle \) be an IOLTS with state-space \( S_{\equiv} \) consisting of \( \equiv \)-equivalence class of \( S \) and \( \rightarrow_{L} \) defined by:

- if \( x \in L \) (with \( L = I \cup U \)) and \( s \xrightarrow{x} t \), then \( [s]_{\equiv} \xrightarrow{x} [t]_{\equiv} \).
- if \( x \in \delta_{\equiv} \) and \( s \neq t \), then \( [s]_{\equiv} \xrightarrow{\sigma} [t]_{\equiv} \).

then

1. \( [s]_{\equiv} \xrightarrow{\varepsilon} [t]_{\equiv} \) implies \( t \equiv s \).
2. \( \forall s \in S \cdot \text{traces}(s) = \text{Straces}_{\equiv}([s]_{\equiv}). \)
3. \( \forall s \in S \cdot \forall \sigma \in L_{u1}^* \cdot \text{out}(s \text{ after } \sigma) = \text{out}([s]_\equiv \text{ after } \sigma) \).

**Proof.**

1. The proof proceeds by induction on the number of \( \tau \)-transition leading to \( s' \in L \). For the base case, we assume that the number of \( \tau \)-transition is 0. Therefore, \( [s]_\equiv \xrightarrow{\epsilon} [t]_\equiv \) implies \( [s]_\equiv = [t]_\equiv \). It is concluded from \( [s]_\equiv = [t]_\equiv \) that \( t \subseteq s \). We assume for the induction step that the above thesis holds for the sequence consisting of \( n - 1 \) \( \tau \)-transition and the number of \( \tau \)-transition leading to \( \xrightarrow{\tau} \) is \( n \).

Thus, there exists state \( [t']_\equiv \) such that it is reachable by \( n - 1 \) \( \tau \)-transition from \( [s]_\equiv \) and \( [t']_\equiv \xrightarrow{\tau} [t]_\equiv \). We know from constructing rules of IOLTS \( L \) that \( \tau \)-transition is taken only under inference rule \( B.1 \). We consider two cases; \( n = 1 \) and \( n > 1 \).

We first, suppose that \( n = 1 \); thus \( s = t' \). Therefore, \( t \) is reachable by a \( \delta_s \)-transition from \( s \) where \( x \in \mathcal{U} \). With respect to quiescence-reducible property, it is obtained that \( t \subseteq s \). Then, as the second case, we suppose that \( n > 1 \). Therefore, \( t' \) is reachable by a \( \delta_{t'} \)-transition where \( x \in \mathcal{U} \). We also conclude from \( [t']_\equiv \xrightarrow{\tau} [t]_\equiv \) that there exists \( y \in \mathcal{U} \) such that \( t' \xrightarrow{\delta_y} t \) and \( t \neq t' \). We distinguish two cases; \( x \subseteq y \) and \( y \subseteq x \).

- We suppose that \( y \subseteq x \). Because \( M \) is a stable extended suspension it respects stability condition. Therefore, \( \text{traces}(t) = \text{traces}(t') \) is resulted. Hence, \( t = t' \) is clearly obtained which leads to \( t \subseteq t' \).
- We suppose that \( x \subseteq y \). Because \( M \) is a stable extended suspension traces it respects quiescence-reducible condition. Therefore, \( \text{traces}(t) \subseteq \text{traces}(t') \) is resulted. Hence, \( t \subseteq t' \) is clear.

Following induction hypothesis, \( t' \subseteq t \) is concluded. The last two results lead to \( \text{traces}(t) \subseteq \text{traces}(t') \subseteq \text{traces}(s) \). Thus, \( t \subseteq s \).

2. We define \( \text{traces}(s)_n \) as the set of the traces with the length of \( n \) starting at state \( s \). Accordingly, \( \text{Straces}_{\mathcal{U}}([s]_\equiv)_n \) denotes the extended suspension traces of state \( [s]_\equiv \in [S]_\equiv \) with the length of \( n \). It is obvious that \( \text{traces}(s) = \bigcup_{n \in \mathbb{N}} \text{traces}(s)_n \) and \( \text{Straces}_{\mathcal{U}}([s]_\equiv) = \bigcup_{n \in \mathbb{N}} \text{Straces}_{\mathcal{U}}([s]_\equiv)_n \). Thus, we prove the equivalent rephrased statement of the given thesis; \( \forall s \in S, \forall n \in \mathbb{N} \cdot \text{traces}(s)_n = \text{Straces}_{\mathcal{U}}([s]_\equiv)_n \). The proof is given by induction on number \( n \).

- **Base case.** For the base case, we assume that \( n = 0 \). Clearly, \( \text{traces}(s)_0 = \text{Straces}_{\mathcal{U}}([s]_\equiv)_0 \) because the only member of both set is the empty sequence, i.e., \( \text{traces}(s)_0 = \text{Straces}_{\mathcal{U}}([s]_\equiv)_0 = \{\epsilon\} \).
- **Induction step.** We assume for the induction step that the given thesis holds for sequences of length \( m \) and that \( n = m + 1 \). We show that \( \text{traces}(s)_m \subseteq \text{Straces}_{\mathcal{U}}([s]_\equiv)_m \) and \( \text{Straces}_{\mathcal{U}}([s]_\equiv)_m \subseteq \text{traces}(s)_m \).
  
  - We prove that \( \text{traces}(s)_m \subseteq \text{Straces}_{\mathcal{U}}([s]_\equiv)_m \).
  
  Consider an arbitrary member of \( \text{traces}(s)_m \), namely \( \sigma \). We take \( \sigma = a\sigma' \) with \( a \in L_{\mathcal{U}} \) and \( \sigma' \in L_{\mathcal{U}1}^* \); thus the length of \( \sigma' \) is \( m \). Therefore, there exists \( s' \in S \) such that \( s \xrightarrow{a} s' \) and \( \sigma' \in \text{traces}(s')_m \). We first show that
We prove that \( a \in \text{Straces}_u([s]_{/\equiv}) \) and also \([s']_{/\equiv} \in ([s]_{/\equiv} \text{ after } a)\). In this regard, we distinguish two cases; \( a \in L \) and \( a = \delta_x \) with \( x \in \Sigma \).

* We suppose that \( a \in L \). Following inference rules defined in Lemma B.1, \([s]_{/\equiv} \overset{a}{\rightarrow}_L [s']_{/\equiv}\) is obtained; \( a \in \text{Straces}_u([s]_{/\equiv}) \). Therefore, \([s']_{/\equiv} \in ([s]_{/\equiv} \text{ after } a)\) is deduced.

* We suppose that \( a = \delta_x \) with \( x \in \Sigma \). Following inference rules defined in Lemma B.1, \([s]_{/\equiv} \overset{\epsilon}{\Rightarrow}_L [s']_{/\equiv}\) is obtained. It is deduced from anomaly-freeness that \( x \not\in \text{init}(s')\). Hence, it is concluded from inference rules defined in B.1 that \( \text{init}([s']_{/\equiv}) \cap x = \emptyset \) as well. Therefore, \( \delta_x \in \text{Straces}_u([s']_{/\equiv}) \) is obtained. Consequently, we conclude that \( \delta_x \in \text{Straces}_u([s]_{/\equiv}) \) and \([s']_{/\equiv} \in ([s]_{/\equiv} \text{ after } \delta_x)\).

We know that \([([s]_{/\equiv} \text{ after } a) = \{[t]_{/\equiv} \mid [s]_{/\equiv} \overset{\epsilon}{\Rightarrow}_L [t']_{/\equiv} \overset{a}{\rightarrow}_L [t]_{/\equiv}\}\) following Lemma B.1, we find that \( \forall [t]_{/\equiv} \in ([s]_{/\equiv} \text{ after } a) \forall [t']_{/\equiv} \in ([s]_{/\equiv} \text{ after } a) \cdot t' \equiv t \) implies \( t' \equiv s \), which consequently results in \( (t' \text{ after } a) \subseteq (s \text{ after } a) \); thus, \( \forall [t]_{/\equiv} \in ([s]_{/\equiv} \text{ after } a) \cdot t \equiv t' \). Regarding \([s']_{/\equiv} \in ([s]_{/\equiv} \text{ after } a)\), it is obtained that \( \bigcup_{[t]_{/\equiv} \in ([s]_{/\equiv} \text{ after } a)} \text{traces}(t) = \text{traces}(s')(\epsilon)\). Following induction hypothesis, it is obtained that \( \forall [t]_{/\equiv} \in ([s]_{/\equiv} \text{ after } a) \cdot \text{traces}(t) = \text{Straces}_u([t]_{/\equiv})\).

We know that \( \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv} = \bigcup_{[t]_{/\equiv} \in ([s]_{/\equiv} \text{ after } a)} \text{traces}(t) \) is obtained. Hence, \( \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv} = \text{traces}(s')_{/\equiv}\) is deduced from \( \ast \). Following \( \sigma' \in \text{traces}(s')_{/\equiv} \), \( \sigma' \in \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv} \) is deduced and consequently \( \sigma \in \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv} \) is obtained. Hence, \( \text{traces}(s)_{/\equiv} \subseteq \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv} \).

We prove that \( \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv} \subseteq \text{traces}(s)_{/\equiv} \).

Consider an arbitrary member of \( \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv}\), namely \( \sigma \). We take \( \sigma = a\sigma' \) with \( a \in L_\Sigma \) and \( \sigma' \in L_\Sigma^* \); thus the length of \( \sigma' \) is \( m \) and consequently, \( \sigma' \in \text{Straces}_u([s]_{/\equiv} \text{ after } a)_{/\equiv}\). We know that \([s]_{/\equiv} \text{ after } a = ([t]_{/\equiv} \mid [s]_{/\equiv} \overset{\epsilon}{\Rightarrow}_L [t']_{/\equiv} \overset{a}{\rightarrow}_L [t]_{/\equiv}\)\). Consider an arbitrary \([t]_{/\equiv} \) and \([t']_{/\equiv}\) such that \([s]_{/\equiv} \overset{\epsilon}{\Rightarrow}_L [t']_{/\equiv} \overset{a}{\rightarrow}_L [t]_{/\equiv}\). First, we show that \( a \in \text{traces}(s)\). In this regard, we consider two cases; \( a \in L \) and \( a = \delta_x \) where \( x \in \Sigma \).

* We suppose that \( a \in L \). Following inference rule B.1, \([t']_{/\equiv} \overset{\delta_x}{\rightarrow}_L [t]_{/\equiv}\) is possible only if \( t' \overset{a}{\rightarrow} t \). Thus, \( a \in \text{traces}(t') \) is resulted.

* We suppose that \( a = \delta_x \) where \( x \in \Sigma \). It follows from \([t']_{/\equiv} \overset{\delta_x}{\rightarrow}_L [t]_{/\equiv}\) that \([t']_{/\equiv} = [t]_{/\equiv}\) and also \( \text{init}([t']_{/\equiv}) \cap x = \emptyset \). The last result leads to \( x \not\in \text{init}(t') \) because otherwise, inference rule B.1 eventuates in \( \text{init}([t']_{/\equiv}) \cap x \neq \emptyset \). Consequently, we deduce from Non-blocking property of \( t' \) that \( \delta_x \in \text{init}(t') \). Thus, \( \delta_x \in \text{traces}(t') \) is clearly obtained.

Therefore, it is proven that \( a \in \text{traces}(t') \). Following the first item of
Chapter B. Proofs of Formal Results in Chapter 7

Lemma B.1, \( t' \subseteq s \) is concluded. The last two observations result in \( a \in \text{traces}(s) \). The remainder of the proof proceeds with the similar lines of reasoning in the previous case.

3. It is directly resulted from the previous item.

\[ \square \]

Now, the proof of Theorem 7.14 is given below.

**Proof of theorem 7.14.** Consider IOLTS \( \langle S, I, U, \rightarrow_L, [\bar{s}]_\equiv \rangle \) which is derived from ESA \( \bar{s} \) with respect to Lemma B.1. We prove IOLTS \( [\bar{s}]_\equiv \) meets both required conditions.

1. It is a direct result of the second item in Lemma B.1.

2. We prove that \( \bar{r} \text{ ioco}_p \bar{s} \) implies \( \bar{r} \text{ ioco}_p [\bar{s}]_\equiv \) where \( F \subseteq L_{U_1}^\prime \). The proof is given by contraposition. We assume \( \bar{r} \text{ ioco}_p [\bar{s}]_\equiv \) so that \( \exists \sigma \in F \cap \text{traces}_U([\bar{s}]_\equiv) \) such that \( x \in \text{out}(\bar{r} \text{ after } \sigma) \) and \( x \not\in \text{out}([\bar{s}]_\equiv \text{ after } \sigma) \) where \( x \in L_{U_1} \). Following Lemma B.1, it is obtained that \( x \not\in \text{out}(\bar{s} \text{ after } \sigma) \) while \( \sigma \in \text{traces}(\bar{s}) \). Therefore, \( \bar{r} \text{ ioco}_p \bar{s} \) is resulted. Then, We prove that \( \bar{r} \text{ ioco}_p [\bar{s}]_\equiv \) implies \( \bar{r} \text{ ioco}_p \bar{s} \). The proof is identical to the proof of the previous item.

\[ \square \]

**Remark B.2.** Hereon, we will write \( L^\delta \) for the set \( I \cup U \cup \{ \delta_U \} \).

**Lemma 7.29.** Let IOLTS \( \bar{s} \) be a specification, and let IOTS \( \bar{e} \) be an environment. Let the behavior of IOTS \( \bar{e} \) be included in the behavior of the given specification \( \bar{s} \), i.e., \( \bar{e} \text{ incl } \bar{s} \), and let \( \sigma \in (L_e \cup L \cup \{ \delta_{(U \cup U_1)} \})^* \) such that

- \( \sigma_{4_L} \in \text{traces}(\bar{s}/\bar{e}) \).
- \( \sigma_{4_{L_e}} \in \text{traces}(\bar{e}) \).
- \( \sigma \) ends in \( (L \cup \{ \delta_{(U \cup U_1)} \}) \).

Then for all \( \sigma' \in \text{traces}(\bar{s}) \) satisfying \( \sigma' = \sigma_{4_{(L_e \cup U_1)}} \) we have:

\( (\bar{s} \text{ after } \sigma', \bar{e} \text{ after } \sigma_{4_{(L_e \cup U_1)}}) = (\bar{s}/\bar{e} \text{ after } \sigma_{4_{(L_e \cup U_1)}})^{-1} \)

**Proof.** The proof proceeds by induction on the length of \( \sigma_{4_{L_e}} \).

- **Base case.** We assume for the base case that \( \sigma_{4_{L_e}} = e \); thus \( \sigma \in (L_e \setminus L_v)^* \). Clearly, \( \sigma = e \) is obtained because \( \sigma \) ends in \( L \cup \{ \delta_{(U \cup U_1)} \} \). Therefore, \( \sigma' \) is empty due to \( \sigma' = e_{L_e} \). Following Definition 7.26, it is clear that \( [\bar{s} \text{ after } \sigma', e \text{ after } \sigma_{4_{L_e}}] \in (\bar{s}/\bar{e} \text{ after } \sigma_{4_{L_e}})^{-1} \) for \( \sigma = \sigma' = e \).

- **Induction step.** We assume for the induction step that the above thesis holds for all sequences ending in \( (L \cup \{ \delta_{(U \cup U_1)} \}) \), the length of projection on \( L \) of which is less than \( n \), and that the length of \( \sigma_{4_{L_e}} \) is \( n \) whereas \( \sigma \) ends in \( (L \cup \{ \delta_{(U \cup U_1)} \}) \). We chop \( \sigma \) into two parts, namely \( \rho_1 \) and \( \rho_2 \) such that the length of \( \rho_{1_{L_e}} \) is \( n-1 \) and it ends

170
in \((L \cup \{\delta_{1(U \cup U_1)}\})\). Consequently, \(\rho_2\) has the structure like \(\rho_e x\) where \(x \in (L \cup \{\delta_s\})\) and \(\rho_2' \in (L_e \setminus L)^*\). Furthermore, we conclude from \(\sigma_{1/e}^*\) and \(\sigma_{1/e}^\prime\) belong to traces \(\bar{e}\) and \(\bar{s}/\bar{e}\) respectively that \(\rho_{21_{\bar{e}x}} = \rho_{\bar{e}} x\) and also \((\sigma_{21_{\bar{e}x}} = x) \in \hat{\text{traces}}(\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}})\). We take an arbitrary sequence of traces(\(\bar{s}\)), namely \(\sigma^*\) such that \(\sigma^* = \sigma_{1/e}^*\). Regarding \(\rho_1\) and \(\rho_2\), we cut \(\sigma^*\) into two parts, namely \(\rho_1^*\) and \(\rho_2^*\), i.e., \(\sigma^* = \rho_1^* \rho_2^*\), such that \(\rho_1^* = \rho_{11_{\bar{e}x}}\) and \(\rho_2^* = \rho_{21_{\bar{e}x}}\). Clearly, \(\rho_1^*\) is a member of traces(\(\bar{s}\)), because \(\rho_1^* \rho_2^* \in \hat{\text{traces}}(\bar{s})\) and consequently, \(\rho_2^* \in \hat{\text{traces}}(\bar{s}\text{ after } \rho_1^*)\). Following induction hypothesis, it is obtained that \([\bar{s}\text{ after } \rho_1^* \bar{s}\text{ after } \rho_{11_{\bar{e}x}}] \in (\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}})^{-1}\). We distinguish four cases based on \(x\).

- We suppose that \(x \in (I \setminus I_e)\). Clearly, it is obtained that \(\rho_{21_{\bar{e}x}} = \rho_e\) and also, \((\rho_2 = \rho_e x) \in L_x^*\); thus \(\rho_2^* = \rho_e x\). Following Definition 7.26 an input transition is taken only under inference rule \(I_1\). Thus, \(\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} x = (\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}})^{-1}\) executes \(x\). We conclude from Definition 7.25 that \([\bar{s}\text{ after } \rho_1^* \rho_e x, \bar{e}\text{ after } \rho_{11_{\bar{e}x}} \rho_e] = \bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} x\). We consequently find that \([\bar{s}\text{ after } \sigma^* \bar{e}\text{ after } \sigma_{1/e}^*] \in \bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*\). Because \(\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^* \in \hat{\text{traces}}(\bar{s})\), we find that \([\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*]^{-1} = \bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*\). Therefore, \([\bar{s}\text{ after } \sigma^* \bar{e}\text{ after } \sigma_{1/e}^*] \in (\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*)^{-1}\).

- We suppose that \(x \in (U \setminus U_v)\). Similar to the previous case, it is clearly resulted that \(\rho_{21_{\bar{e}x}} = \rho_e\) and \((\rho_2 = \rho_e x) \in L_x^*\); thus \(\rho_2^* = \rho_e x\). We conclude from \(x \in \hat{\text{traces}}(\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}})^{-1}\) that \(\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} \in \hat{\text{traces}}(Q_x \times Q_e)\). We know that \(\gamma\)-transition where \(\gamma \in (U \setminus U_v)\) is possible in \(\bar{s}/\bar{e}\) only under inference rule \(U_2\) in Definition 7.26. Therefore, \(\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} x = (\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}})^{-1}\) executes \(x\). It is deduced from Definition 7.25 that \([\bar{s}\text{ after } \rho_1^* \rho_e x, \bar{e}\text{ after } \rho_{11_{\bar{e}x}} \rho_e] \in \bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} x\). Therefore, \([\bar{s}\text{ after } \sigma^* \bar{e}\text{ after } \sigma_{1/e}^*] \in \bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*\). Clearly, \([\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*]^{-1} = \bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*\) because \(\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^* \in \hat{\text{traces}}(\bar{s})\). Therefore, \([\bar{s}\text{ after } \sigma^* \bar{e}\text{ after } \sigma_{1/e}^*] \in (\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*)^{-1}\).

- We suppose that \(x \in L_\nu\). It is clear that \(\rho_{21_{\bar{e}x}} = \rho_e\) and also, \(\rho_2^* = \rho_e\). We conclude from Definition 7.26 that transition \(x\) is taken under inference rule \(U_1\) in the case \(x \in L_\nu\) or inference rule \(U_1\) for \(x \in U_v\). Therefore, \(\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} x = (\bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}})^{-1}\) executes \(x\). We conclude from Definition 7.25 that \([\bar{s}\text{ after } \rho_1^* \rho_e x, \bar{e}\text{ after } \rho_{11_{\bar{e}x}} \rho_e] \in \bar{s}/\bar{e}\text{ after } \rho_{11_{\bar{e}x}} x\). We consequently find that \([\bar{s}\text{ after } \sigma^* \bar{e}\text{ after } \sigma_{1/e}^*] \in \bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*\). Because \(\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^* \in \hat{\text{traces}}(\bar{s})\), we obtain that \([\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*]^{-1} = \bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*\). Therefore, \([\bar{s}\text{ after } \sigma^* \bar{e}\text{ after } \sigma_{1/e}^*] \in (\bar{s}/\bar{e}\text{ after } \sigma_{1/e}^*)^{-1}\).

- We suppose that \(x = \delta_{1(U \cup U_1)}\). We know that \(\rho_{21_{\bar{e}x}} = \rho_e \delta_e\) and also, \(\rho_2^* = \rho_e \delta_e\). We also know that \(\delta\)-transition is taken in \(\bar{s}/\bar{e}\) under inference rule \(\delta_1\). From
Lemma 7.32. Let IOLTS $\bar{\sigma}$ be a specification, and let IOTS$^n \bar{\epsilon}$ be an environment, if

- $\bar{\epsilon}$ incl $\bar{s}$
- $\bar{s}_{/\bar{\epsilon}}$ is a valid quotient automaton (with initial state $\bar{q}$)

then, for every $\sigma \in \text{traces}(\bar{s}_{/\bar{\epsilon}})$, and every $(s, e) \in q^{-1}$ with $q = \bar{q}$ after $\sigma$, we also have,

$$\exists \sigma' \in (L \cup L_e \cup \{\delta_{(U \cup U_1)}\})^* \cdot s = (\Delta(\bar{s}) \text{ after } \sigma'_{\bar{I}_{/\bar{\epsilon}}}) \land (q, e) = (\bar{q} \Delta(\bar{\epsilon}) \text{ after } \sigma')$$

Proof. The proof is given by induction on the length of $\sigma$. We implicitly show $U \cup U_e$ by $O$.

- **Base case.** We assume for the base of the induction that $\sigma = \epsilon$. Following Definition 7.26, it is resulted that the only consisting pair of $(q_0, \text{ after } \epsilon)$ is $(\bar{s}, \bar{\epsilon})$. Clearly, the above thesis holds for $\sigma' = \epsilon$, because $\bar{s} = (\bar{s} \text{ after } \sigma'_{\bar{I}_{/\bar{\epsilon}}})$ and $q_0 \parallel \bar{\epsilon} = (\bar{q} \parallel \bar{\epsilon})$ after $\sigma'$. Clearly, $\sigma'_{\bar{I}_{/\bar{\epsilon}}} = \sigma = \epsilon$.

- **Induction step.** For the induction step, we assume that the above thesis holds for all sequences with the length of $n-1$ and that the length of $\sigma$ is $n$. We suppose $\sigma = \rho x$ where $\rho \in (L \cup \{\delta\})^*$ and $x \in (L \cup \{\delta\})$. It is deduced from determinism of $\bar{s}_{/\bar{\epsilon}}$ that there exist $q', q \in \bar{Q}$ such that $q_0 \xrightarrow{\rho} q' \xrightarrow{x} q$. Following the induction step, for any pair $(q'_e, q'_{\bar{\epsilon}}) \in q^{-1}$, there exists $\rho' \in (L_e \cup L \cup \{\delta_{(U \cup U_1)}\})^*$ such that $q'_e = (\bar{s} \text{ after } \rho'_{\bar{I}_{/\bar{\epsilon}}})$ and $q' \parallel q'_e = (q_0 \parallel \bar{\epsilon} \text{ after } \rho')$ and also, $\rho'_{\bar{I}_{/\bar{\epsilon}}} = \rho$. We distinguish three cases based on $x$:

  - We suppose that $x \in (L \backslash L_e)$. We know from inference rules $I_1$ and $U_2$ in Definition 7.26 that $q = q^{-1}$ executes $x$. Following Definition 7.25, it is obtained that for any pair $(q_s, q_e) \in q$, there exists a pair $(q'_s, q'_e) \in q^{-1}$ such that for a sequence $\sigma_e \in \text{traces}(q'_s) \cap \text{traces}(q'_e)$, it holds that $(q_s = q'_s \text{ after } \sigma_e)$ and $(q_e = q'_e \text{ after } \sigma_e)$; thus $q_s = \bar{s}$ after $(\rho' \sigma_e x)_{\bar{I}_{/\bar{\epsilon}}}$. Regarding Definition 7.15, we know that $q' \parallel q'_e \xrightarrow{\sigma_e} q' \parallel q_e \xrightarrow{x} q \parallel q_e$. Furthermore, $\rho' \sigma_e x_{\bar{I}_{/\bar{\epsilon}}} = \rho x = \sigma$. Hence, the above thesis holds for $\sigma' = \rho' \sigma_e x$.

  - We suppose that $x \in (L_e)$. We know from inference rule $I_1$ and $U_2$ in Definition 7.26 that $q = q^{-1}$ executes $x$. Following Definition 7.25, it is obtained that for any pair $(q_s, q_e) \in q$, there exists a pair such as $(q'_s, q'_e) \in q^{-1}$ such that for a sequence $\sigma_e \in \text{traces}(q'_s) \cap \text{traces}(q'_e)$, it holds that $(q_s = q'_s \text{ after } \sigma_e)$ and $(q_e = q'_e \text{ after } \sigma_e)$; thus, $q_s = \bar{s}$ after $(\rho' \sigma_e x)_{\bar{I}_{/\bar{\epsilon}}}$. Following Definition 7.15, it is clear that $q' \parallel q'_e \xrightarrow{\sigma_e} q' \parallel (q'_e \text{ after } \sigma_e) \xrightarrow{x} q \parallel q_e$. Furthermore, $\rho' \sigma_e x_{\bar{I}_{/\bar{\epsilon}}} = \rho x = \sigma$. Hence, the above thesis holds for $\sigma' = \rho' \sigma_e x$.  

172
We suppose that $x = \delta$. Regarding Definition 7.26, a $\delta$-labeled transition is taken only under rule $\delta_1$ in $S/\bar{e}$. It follows from rule $\delta_1$ that $q = q^{-1}$ executes $x$. Following Definition 7.25, it is concluded that for any pair $(q_s, q_e) \in q^{-1}$, there exists a pair such as $(q_s', q_e') \in q'_s$ such that for a sequence $\sigma_e \in \text{traces}(q_e') \cap \text{traces}(q_e)$, it holds that $(q_s, q_e) = (q_s' \text{ after } \sigma_e \delta_s, M_e' \text{ after } \sigma_e \delta_e)$. Furthermore, $\rho' \sigma_e \delta_{O_l} = \rho \delta = \sigma$. Therefore, the given thesis holds for $\sigma' = \rho \sigma_e \delta_{O_l}$. 

We recall Proposition 7.33 from the body of the thesis.

**Proposition 7.33.** Let IOLTS $\bar{s}$ be a specification, and let IOTS $\bar{e}$ be an environment, if

- $\bar{e} \text{ incl } \bar{s}$, and
- $\bar{s}_{/\bar{e}}$ is a valid quotient automaton

then

$$\forall c \in \text{IOTS}(I, U) \bullet \text{hide}[L_v] \text{ in } c \| \bar{e} \text{ ioco } \bar{s} \text{ implies } c \text{ ioco } \bar{s}_{/\bar{e}}$$

The correctness of the above proposition relies on the following lemma.

**Lemma B.3.** Let IOLTS $\langle Q_s, I_s, U_s, \rightarrow, s \rangle$ be a given specification, and let LTS $\langle Q'_s, I_s \cup U_s \cup \{\delta_s\}, \rightarrow', \bar{s} \rangle$ be the suspension automaton derived from IOLTS $s$ regarding Definition 7.10. Then, $\forall \sigma \in (L \cup \{\delta_s\})^* \bullet \text{out}(s \text{ after } \sigma) = \text{out}(\bar{s} \text{ after } \sigma)$.

**Proof.** Obviously, Tretmans’ transformation [Tre96c] to convert IOLTS $s$ to an SA is a specialized version of transformation presented in Definition 7.10 for $U = \{U_s\}$. Therefore, SA $\bar{s}$ has all properties of SA automaton derived from Tretmans’ transformation. So, regarding Proposition 4.17 in [Tre96c] it is obtained that $\forall \sigma \in (L \cup \{\delta_s\})^* \bullet \text{out}(s \text{ after } \sigma) = \text{out}(\bar{s} \text{ after } \sigma)$. 

\[\square\]
Appendix C

UPPAAL Models

This chapter presents some representative UPPAAL models that have been used in the case study of Chapter 3. The behavior of the EFT switch in this case study is specified in terms of a number of transaction flows. Modeling all of these flows into one model would hamper readability and maintainability. Therefore, the specification of the EFT switch is broken into several automata, each modeling the behavior of EFT switch in a specific transaction flow. The same approach is taken in modeling the environment. In the remainder of this section, the models of a few financial transactions including Purchase transaction (with its reversal transaction) and Refund transaction (with its reversal transaction) are given.

The labels of transitions in the UPPAAL models presented below are encoded as follows: Labels are formatted as $a\_b\_c(_d)_e\_f(_r)$ where

- $a$ and $b$ indicate the sender and the receiver of a message, respectively. The following abbreviations are used,
  - p: POS
  - sw: Switch
  - cs: Core
- $c$ indicates the type of transaction. The following abbreviations are used,
  - p: Purchase
  - rev: Reversal
  - ref: Refund

  For reversal transactions, $d$ indicates the type of the original transaction.

- $e$ indicates the role (sender or receiver) of the switch in financial messages.

- $f$ indicates the type (request or response) of financial messages.

- $r$ indicates the result value of a response message. This field appears only if the corresponding request has not been successful.
For example, according to the above encoding, "p_sw_pur_rcv_rq" represents a purchase request sent to the switch by a POS terminal, "sw_p_pur_snd_rs" represents a purchase response sent to a POS terminal by the switch, and "cs_sw_rev_pur_rcv_rs_91" represents the response message of an unsuccessful purchase reversal transaction between the switch and the core.

In the remainder of this chapter the UPPAAL models of purchase transaction and refund transaction in the EFT system are given. Other financial transactions have been modeled in a very similar way to these two transactions.

A simplified snapshot of part of the behavior of states s_1 and s_2 in Figure C.1 is shown in Figure 6.1 in Chapter 6. After forwarding the received purchase request to the core (sw_cs_pur_snd_rq) at state s_1, the EFT switch reaches state s_2. At state s_2, the corresponding purchase response can be received from the core (cs_sw_pur_rcv_rs), or after time-out elapsed the response message with unsuccessful result (91) is sent to the corresponding POS terminal (sw_p_pur_snd_rs_91) and immediately after that the reversal request for the purchase transaction is sent to the core (sw_cs_rev_pur_snd_rq). As depicted in C.1, at states s_1 and s_2 there are race situations between input actions and output actions, which can be problematic in testing in the asynchronous setting as explained in this thesis.
Figure C.1: The model of a purchase transaction in the EFT switch
Figure C.2: The model of a purchase transaction of (a) a POS terminal, (b) the core
Figure C.3: The model of the lossy communication channel in a purchase transaction from (a) a POS terminal to the switch, (b) the switch to a POS terminal
Figure C.4: The model of the lossy communication channel in a purchase transaction from (a) the switch to the core, (b) the core to the switch
Figure C.5: The model of a refund transaction in the EFT switch
Figure C.6: The model of a purchase transaction of (a) a POS terminal, (b) the core
Figure C.7: The model of the lossy communication channel in a refund transaction from (a) a POS terminal to the switch, (b) the switch to a POS terminal
Figure C.8: The model of the lossy communication channel in a refund transaction from (a) the switch to the core, (b) the core to the switch.
Bibliography


BIBLIOGRAPHY


Summary

Improving Input-Output Conformance Testing Theories

Software quality has become an increasingly important concern in software development, as a consequence of the ever-ascending trend of using software systems in critical systems. Subsequently, testing as a technique for establishing a certain level of software quality has received much attention in the past decades. Model-based testing is a structured approach to testing. Using model-based testing the process of generating test-cases and predicting the correct outcome of test-cases (the so-called oracle problem) can be mechanized. By using rigorous models for system behavior, model-based testing is formalized in terms of a mathematical notion of conformance.

Input-output conformance (ioco) is a widely-studied and commonly-used conformance relation. The ioco relation is initially developed for the synchronous communication setting. It is also well-known that the ioco relation does not have the compositionality property. These characteristics can be an obstacle in testing of concurrent systems with the ioco relation. Concurrent systems are composed of interacting components which often communicate asynchronously with their environment. This thesis studies the characteristics of the ioco relation to make it suitable for testing of concurrent systems. In particular, we address the following topics.

- We look into the ioco relation from the computational point of view. It is shown that the problem of checking the ioco relation is PSPACE-COMPLETE. We present a polynomial time algorithm for deciding the ioco relation in a confined setting, namely for deterministic specifications.

- We investigate the testing power of the ioco relation in comparison with some other conformance relations in the literature. Originally, the ioco relation is developed for systems modeled as an input-output labeled transition system. We compare the ioco relation with the conformance relation introduced for internal-choice labeled transition system, and show that they are not equal.

- A comprehensive study of testing with the ioco relation in the asynchronous setting is conducted. In this regard, we characterize different subclasses of implementations and specifications with the same observational behavior, with respect to the ioco relation, in presence and absence of buffers. We show that the state-space explosion problem, caused by adding buffers to models in asynchronous setting,
can be sidestepped by restricting either implementations or specifications to the presented subclasses.

- We study the property of decomposability of a specification with respect to the ioco relation. Decomposability allows for using the ioco relation in component-based and product-line software systems, where it is assumed that some parts of the system under development are reused components, which we collectively call platform. We define the property of decomposability on labeled transition systems formally, and consequently propose an algorithm to check that property. In this regard, we introduce the quotient automaton which is derived from a given specification with respect to the platform. We investigate the necessary and sufficient conditions for the validity of the quotient automaton, and prove that the conformance of any new component to the quotient automaton ensures the correctness of the whole system built up of the new component and the platform.
Curriculum Vitae

Neda Noroozi was born on 23 May 1982 in Mashhad, Iran. After obtaining her Bachelor of Science degree in computer engineering in 2004 at Sharif University of Technology in Tehran, Iran, she studied Software engineering at the same University. She received her Master of Science degree with honors in September 2006. Her Master's thesis was titled "Validation and verification of reconfigurable software systems: a method based on adaptive software architecture". From March 2010 she started a PhD project at Formal system analysis Group, Department of Mathematics and Computer Science, Eindhoven University of Technology, the Netherlands of which the results are presented in this dissertation.

As of June 2014 she started working on industrial and applied research projects in the field of model-based testing at Nspyre B.V in Eindhoven, the Netherlands.
Titles in the IPA Dissertation Series since 2008


N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06


I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Clephas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10


M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12


E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16


M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18


E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23


H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27


M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02


M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06


J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11


M.R. Czenko. TuLiP - Reshaping Trust Management. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. Clocks, Dice and Processes. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17


R.S.S. O'Connor. Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. Improved Verification Methods for Concurrent Systems. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. Diagnosis, Synthesis and Analysis of Probabilistic Models. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P Kwisthout. The Computational Complexity of Probabilistic Networks. Faculty of Science, UU. 2009-23

T.K. Cocx. Algorithmic Tools for Data-Oriented Law Enforcement. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. Embedded Compilers. Faculty of Science, UU. 2009-25

M.A.C. Dekker. Flexible Access Control for Dynamic Collaborative Environments. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.E.J. Laros. Metrics and Visualisation for Crime Analysis and Genomics. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Booger. Focusing Automatic Code Inspections. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. Model Checking Nondeterministic and Randomly Timed Systems. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. Termination and Productivity. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. Graph-Based Specification and Verification for Aspect-Oriented Languages. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. Epistemic Modelling and Protocol Dynamics. Faculty of Science, UvA. 2010-05

J.K. Berendsen. Abstraction, Prices and Probability in Model Checking Timed Automata. Faculty of Science, Mathematics and Computer Science, RU. 2010-06
A. Nugroho. The Effects of UML Modeling on the Quality of Software. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. Kleene Coalgebra. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. Formal Models for Component Connectors. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services. Faculty of Mathematics and Natural Sciences, UL. 2010-11


B.J. Arnoldus. An Illumination of the Template Enigma: Software Code Generation with Templates. Faculty of Mathematics and Computer Science, TU/e. 2011-02


L. Astefanoaei. An Executable Theory of Multi-Agent Systems Refinement. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. Synchronous coordination of distributed components. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Morali. IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. On changing models in Model-Based Testing. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. Reconfigurable Component Connectors. Faculty of Mathematics and Natural Sciences, UL. 2011-08


M. Atif. Formal Modeling and Verification of Distributed Failure Detectors. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. From Computability to Executability – A process-theoretic view on automata theory. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. Configuration management for models: Generic methods for model comparison and model co-evolution. Faculty of Mathematics and Computer Science, TU/e. 2011-12


C.P. Tsirogiannis. Analysis of Flow and Visibility on Triangulated Terrains. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. Stochastic Models for Quality of Service of Component Connectors. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. Capturing and Exploiting Abstract Views of States in OO Verification. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.E. van Amstel. Assessing and Improving the Quality of Model Transformations. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. Towards Correct Programs in Practice. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. Ambiguity Detection for Programming Language Grammars. Faculty of Science, UvA. 2011-21

M. Izadi. Model Checking of Component Connectors. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. Building Blocks for Language Workbenches. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. Modelling and Analysis of Real-Time Coordination Patterns. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. Spiking Neural P Systems. Faculty of Mathematics and Natural Sciences, UL. 2011-25


A. Middelkoop. Inference of Program Properties with Attribute Grammars, Revisited. Faculty of Science, UU. 2012-02

Z. Hemel. Methods and Techniques for the Design and Implementation of Domain-Specific Languages. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03


S. Sedghi. Towards Provably Secure Efficiently Searchable Encryption. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference. Faculty of Science, Mathematics and Computer Science, RU. 2012-06


H. Rahmani. Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. Software Language Evolution. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. From Napkin Sketches to Reliable Software. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P. M. Stappers. Bridging Formal Models – An Engineering Perspective. Faculty of Mathematics and Computer Science, TU/e. 2012-12
W. Heijstek. Software Architecture Design in Global and Model-Centric Software Development. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. Higher Order Termination. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. Formal Development of Control Software in the Medical Systems Domain. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. Compositional Synthesis of Safety Controllers. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16


B. Lijnse. TOP to the Rescue – Task-Oriented Programming for Incident Response Applications. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. Outsmarting Smart Cards. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. Test Suite Comprehension for Modular and Dynamic Systems. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. Interactive mathematical documents: creation and presentation. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. Composition and synchronization of real-time components upon one processor. Faculty of Mathematics and Computer Science, TU/e. 2013-08


M.J.M. Roeloffzen. Kinetic Data Structures in the Black-Box Model. Faculty of Mathematics and Computer Science, TU/e. 2013-14


C. de Gouw. Combining Monitoring with Run-timeAssertion Checking. Faculty of Mathematics and Natural Sciences, UL. 2013-17


T.M. Ngo. Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. Scalable Multi-Core Model Checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. Coalgebraic Characterizations of Automata-Theoretic Classes. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. Similarity Measures and Algorithms for Cartographic Schematization. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. JTorX: Exploring Model-Based Testing. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. Domain Specific Languages and their Type Systems. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. Social Aspects of Collaboration in Online Software Communities. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. Tomte: Bridging the Gap between Active Learning and Real-World Systems. Faculty of Science, Mathematics and Computer Science, RU. 2014-12