Dissertation

Automated Software Conformance Testing

Martin Weiglhofer

Graz, 2009

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: Dr. ir. Gerrit Jan Tretmans, Associate Professor

Abstract (English)

Software has found its way into our everyday lives and as a consequence the impact of software failures can range from just being inconvenient to being life-threatening in the worst case. Consequently, software quality is a major concern in the development of modern software systems. Testing is considered as a practical technique for finding bugs in computer programs. An important aspect to be considered is conformance testing, which is used to validate the correctness of an implementation with respect to its model. In such a setting a conformance relation formalizes the notion of correctness. One commonly used conformance relation is the input-output conformance (ioco) relation. This thesis aims to provide theories and techniques that facilitate the application of ioco testing in an industrial context.

Originally, the input-output conformance relation was formulated over labeled transition systems and so far some of the underlying assumptions have only been stated on an informal basis. This thesis reflects the input-output conformance relation to reactive processes of the Unifying Theories of Programming framework. Thereby, all assumptions of ioco are formalized. Furthermore, this thesis shows how some of these assumptions can be relaxed.

Knowing the pros and cons of a particular testing technique leads to the next step in conformance testing: the selection of an appropriate set of test cases. One promising test case selection technique in the context of ioco testing is the use of formal test objectives, i.e. test purposes. However, nowadays such test purposes are usually developed by hand which makes the quality of the resulting test suite dependent on the skills of the tester. This thesis investigates techniques that support test engineers in complementing manually generated test purposes and by automatically deriving test purposes.

Once a set of test cases has been generated and executed a lot of time is spent on the analysis of the test execution results. This thesis describes an approach that intends to assist test engineers in grouping test cases with respect to detected failures. By analyzing only one test case per group the time and effort needed for the test result analysis can be significantly reduced.

To sum up, this thesis makes input-output conformance testing more amenable to practical implementation.

Zusammenfassung (German)

Software hat längst Einzug in unser tägliches Leben gehalten. Aus diesem Grunde können Fehler in Software Systemen kleine Unbequemlichkeiten sein oder aber im schlimmsten Fall sogar lebensbedrohlich werden. Die Qualität von Software muss daher ein wichtiges Anliegen in der Software Entwicklung sein. Software Testen ist eine, in der Industrie akzeptierte, praxisnahe Technik. Ein wichtiger Aspekt ist die Konformität einer Implementierung in Bezug auf das Modell des implementierten Systems. Im Testen auf Konformität formalisiert eine Konformitätsrelation den Begriff der Korrektheit. Eine gängige Relation ist die Input-Output Konformitätsrelation (ioco). In dieser Arbeit werden Theorien und Techniken vorgestellt, die die Anwendung von Input-Output Konformtätstesten im industriellen Kontext erleichtern.

Die ioco Relation wurde ursprünglich auf Basis von Transitionssystemen formuliert und einige der zugrunde liegenden Annahmen wurden nur informal getroffen. In dieser Arbeit wird die ioco Relation auf Basis reaktiver Prozesse der "Unifying Theories of Programming" wiedergegeben. Dabei werden alle Annahmen formalisiert. Im Weiteren wird gezeigt wie einige dieser Annahmen abgeschwächt werden können.

Kennt man die Vor- und Nachteile einer bestimmten Technik, ist der nächste Schritt das Auswählen von angemessenen Testfällen. Ein vielversprechender Ansatz ist die Verwendung von formalen Testzielen. Allerdings werden diese formalen Testziele heutzutage händisch entwickelt und daher ist das Ergebnis der Testfallgenerierung sehr von den Fähigkeiten des Testingenieurs abhängig. In dieser Arbeit werden Techniken untersucht die diesen manuellen Prozess unterstützen indem händische Testziele vervollständigt und Testziele automatisch abgeleitet werden.

Durch die Verwendung von Modellen ist die Anzahl der erstellten Testfälle oft sehr groß und daher der Zeitaufwand für die Analyse der Testfallausführung ebenfalls entsprechend hoch. In dieser Arbeit wird ein Ansatz beschrieben der den Testingenieur bei der Analyse der Testfallausführung unterstützt. Die Testfälle werden nach der Ausführung entsprechend der detektierten Fehler gruppiert. Danach muss nur mehr ein Testfall pro Gruppe analysiert werden wodurch die Analysezeit erheblich reduziert wird.

Zusammenfassend kann festgehalten werden, dass es das Ziel dieser Arbeit ist, Konformitätstesten zugänglicher für die praktische Anwendung zu machen.

Acknowledgement

I would like to thank many people who gave me great support during this thesis. First of all, I am grateful to my adviser Prof Franz Wotawa for his support and for his valuable comments on this work. I would also like to thank Jan Tretmans for reviewing this thesis.

Thanks to Gordon Fraser for fruitful discussions and for having an open chat window for me all the time. I am indebted to Bernhard Aichernig who pointed me to the Unifying Theories of Programming and who had many valuable comments on my work. Special thanks go to Stefan Galler, who has become a good friend during my studies and we shared an office. He positively influenced my work and had always time for valuable discussions. I'm grateful to my colleagues Willibald Krenn and Harald Brandl for fruitful discussions. I would also like to thank Karin Richter for proof-reading.

Of course, thanks to my parents for making my studies possible and for the assistance during all the years. Finally, I would like to thank my wife Sonja for the support, the encouragement and the patience provided during my years of study. She always put me up during the hard times.

The thesis was partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

Martin Weiglhofer
Graz, Austria, 2009

Statutory Declaration

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

Place	Date	Signature

Eidesstattliche Erklärung

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

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Software development is a complex and error-prone task. Failures in safety-critical applications may be life-threatening. Previous software bugs have dramatically shown how software effects our daily life and how severe software errors can became. Wired News (Garfinkel, 2005) published the following list as the ten worst software bugs:

1962 - Mariner I space probe Due to an improper translation of a formula into computer code the rocket diverted from its intended launch path and had to be destroyed.

1982 – Soviet gas pipeline A bug, allegedly injected by operatives working for the CIA, in the trans-Siberian gas pipeline computer system caused the explosion of the pipeline.

1985-1987 – Therac-25 medical accelerator At least five patients died because of a race condition in a radiation therapy device.

1988 – Buffer overflow in Berkeley Unix finger daemon The first internet worm made use of this buffer overflow.

1988-1996 – Kerberos Random Number Generator The lack of a proper random seed caused systems that relied on Kerberos for authentication to be trivially hacked.

1990 – AT&T Network Outage A bug in the control software of AT&T's #4ESS long distance switches left approximately 60 thousand people without long distance service for nine hours.

1993 – Intel Pentium floating point divide A silicon error in the Intel Pentium's floating point unit cost Intel $475 million.

1996 – The Ping of Death Missing sanity checks in the IP fragmentation reassembly code made it possible to crash a wide variety of operating systems by malformed "ping" packets.

1996 – Ariane 5 Flight 501 The Ariane 5 rocket was destroyed 40 seconds after launch because the rocket veered off its flight path due to an arithmetic overflow caused by an incorrect conversion from 64-bit numbers to 16-bit numbers.

2000 – National Cancer Institute, Panama City At least eight patients died because of a miscalculation of the proper dosage of radiation for their radiation therapy.

This list is an appalling cue that software systems need to undergo a thorough testing and validation phase. However, it is estimated that on average testing and validation take up 50% of a software project budget (Myers, 1979; Alberts, 1976). Therefore, software engineers need the support of tools, techniques, and theories in order to reduce the number of software failures. While formal proof methods can show the correctness of a software system their applicability is limited to systems of limited size and a full automation is not possible. On the contrary, software testing can be automated and its applicability is not limited to small software systems.

However, as Dijkstra already said “Testing shows the presence, not the absence of bugs” (see Buxton and Randell (1970)). In other words, testing can only be used to gain some confidence in the correctness of the system under test. Only exhaustive testing can show the correctness of the software. Unfortunately, exhaustive testing is infeasible for practical software systems.

Nevertheless, software testing is nowadays a practical technique that is used in industry. As test cases are most often written by hand, testing, if conducted systematically and thoroughly, is a tedious, time-consuming, and error-prone task. Thus there is a great need for formal theories, tools and techniques supporting the automation of software testing in the software development cycle.

However, applying formal testing techniques in practice raises additional challenges that have to be considered. Besides the selection of an appropriate theory there is the problem of test case selection: How can a finite subset of test cases from the possibly infinite set of existing test cases be obtained. Once a set of test cases has been chosen and executed on an implementation the question is which failures have been detected.

Of course there are plenty of other challenges in software testing and an exhaustive list would fill books. This thesis addresses some of the issues when applying formal testing techniques to industrial applications. It was mainly conducted within the “Formal Methods in Software Engineering of Mobile Applications” project of the Softnet competence network*. The project started in 2006 in cooperation with the Institute of Software Technology of the Graz University of Technology, the Kapsch CarrierCom AG, and the Department of Distributed and Multimedia Systems of the University of Vienna as project partners.

The project aim of “Formal Methods in Software Engineering of Mobile Applications” was to apply formal methods to industrial scale systems and to develop techniques that bridge the gap between formal testing in academic research and formal testing of industrial applications. As Kapsch CarrierCom provides telecommunication infrastructure the system under test was an implementation of the Session Initiation Protocol (SIP) Registrar. A SIP Registrar is responsible for maintaining location information of users in an Voice-Over-IP system. As a Voice-Over-IP systems often consists of different entities provided by different vendors, the compliance with the official SIP standard (Rosenberg et al., 2002) is of major importance. This thesis reports on the results obtained within that project.

1.2 Problem Statement

This thesis investigates the problem of using input-output conformance testing to assess the correctness of an implementation with respect to its specification. Applying formal testing techniques in industry is a challenging task in general. Although, a lot of research in the area of input-output conformance testing has brought major advances, there are still many issues that need to be solved. The main issues addressed by this thesis are the following:

Unification The very first step in employing formal testing techniques is the selection of a proper technique that is powerful enough to test all important properties of the system under test. Besides, this technique needs to be as simple as possible in order to avoid unnecessary computational complexity. Automated software testing is a difficult problem. Consequently, different techniques introduce different assumptions in order to cope with particular issues. Unfortunately, the various concepts are described using different formalisms which makes a direct comparison impossible. This raises the need to express the different techniques in a unified framework where all assumptions are formally explained. The basic question is: What are the underlying assumptions of a particular technique and what are the properties of specifications, implementations and test cases?

*`http://www.soft-net.at`

Testing Assumptions Testing theories employ certain assumptions in order to simplify the theories. For example, the input-output conformance testing theory assumes that implementations are input-enabled, i.e. they accept any input at any moment in time, and that test case execution is conducted synchronously. These assumptions are sometimes hard to establish in practice. How can we relax these premises, making input-output conformance testing more amenable to practical implementation?

Test Case Selection Having a proper theory and knowing the strength and weaknesses of this theory raises the question of how to select test cases. How can a finite subset of test cases be obtained from the possibly infinite set of existing test cases. While there are many methods and techniques especially the use of formal test objectives, i.e. the use of test purposes, has shown to be a promising technique. However, these days test purposes are mostly written by hand which makes this method very dependent on the skills of the test engineer. How can we assist a test engineer in the development of formal test objectives?

Test Result Analysis After all, once a set of test cases has been derived and executed there is the question which failures have been detected. As model based testing often leads to large test suites the analysis of the test case execution results may become a tedious and time consuming task. How can we support test engineers in the post analysis of the test case execution?

1.3 Thesis Statement and Contributions

Thesis Statement. The applicability of input-output conformance testing can be improved by formally defining its underlying assumptions, by relaxing some of these, and by assisting test engineers in the design of test objectives and the analysis of test execution results.

Contributions. Parts of the work presented in this thesis have been published in international workshops, conferences, and journals. All of these publications were formally peer reviewed:

- A reformulation of the input-output conformance theory in the Unified Theories of Programming, published in (Weiglhofer and Aichernig, 2008b) and (Weiglhofer and Aichernig, 2008a). The presented denotational semantics gives a precise definition of all assumptions and allowed us to relate refinement and the input-output conformance theory.
- The input-output conformance relations employs some assumptions that limits its practical applicability. These assumptions had been relaxed in (Weiglhofer and Wotawa, 2009a).
- The use of formal test objectives, so called test purposes, has proven to be an efficient way for test case selection. However, the quality of the resulting test suites highly depends on the quality of the test purposes. An approach tackling this issue by creating multiple test cases for a single test purpose has been published in (Fraser, Weiglhofer, and Wotawa, 2008b).
- Test purpose based testing with respect to particular faults, is only applicable to specifications of limited size. An approach to apply fault based test purposes to large specifications has been presented in (Aichernig, Peischl, Weiglhofer, and Wotawa, 2007). Fault based test purposes have been further improved in (Aichernig, Weiglhofer, and Wotawa, 2008), (Weiglhofer and Wotawa, 2008a) and in (Weiglhofer, Aichernig, and Wotawa, 2009).
- An approach using the coverage of LOTOS specifications for the automatic generation of test purposes and for complementing manually designed test purposes has been published in (Fraser, Weiglhofer, and Wotawa, 2008a) and in (Weiglhofer, Fraser, and Wotawa, 2009a).
- Having different test case selection techniques raises the question how these approaches compare. The findings of a comparison were published in (Weiglhofer and Wotawa, 2008b).
- Deriving test cases from formal models leads to abstract test cases that cannot be directly fed into an implementation under test. A rule-based test case execution framework that bridges the gap between the abstract test case and the concrete implementation has been presented in (Peischl, Weiglhofer, and Wotawa, 2007).

- Despite all test case selection techniques model based testing often leads to large test suites. Thus, the manual analysis of the test case execution results becomes a tedious and time-consuming task. An approach for grouping test cases with respect to the detected failure has been presented in (Weiglhofer, Fraser, and Wotawa, 2009b).

1.4 Organization

The remaining chapters of this thesis are organized as follows: Chapter 2 gives an overview of software testing and discuss how this thesis fits into the overall software testing context. Chapter 3 reviews the state of the art in input-output conformance testing and surveys findings related to this thesis.

Chapter 4 recasts the input-output conformance testing theory in the Unifying Theories of Programming framework. Chapter 5 tackles some assumptions regarding the conformance testing theory and shows how these assumptions can be relaxed by imposing certain limitations on the used specifications and implementations.

Chapter 6 introduces three different approaches for assisting a test engineer in the design of formal test objectives, i.e. in the design of test purposes. The presented techniques are based on coverage and on anticipated fault models. Once a set of test cases has been generated and executed, the approach presented in Chapter 7 groups test runs with respect to the detected failures. This technique assists a test engineer in the analysis of the test execution.

Chapter 8 shows the results obtained when applying our techniques to two different protocol specifications. Furthermore, this section discusses how test cases are executed by a generic rule-based rewriting system. Finally, a conclusion is drawn in Chapter 9 and an outlook for further research on the topic is given.

Chapter 2

Software Testing

According to the IEEE Software Engineering Body of Knowledge (Abran and Moore, 2004) testing is “an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program [. . .]”. Myers (1979) defines testing as “the process of executing a program with the intent of finding errors”. According to Ammann and Offutt (2008) testing means “evaluating software by observing its execution”. Utting and Legeard (2007) names testing “the activity of executing a system in order to detect failures”. Whittaker (2000) says that “software testing is the process of executing a software system to determine whether it matches its specification and executes in its indented environment”.

All these definitions show that the inherent nature of software testing is the execution of the implementation under test. Basically, this is the main difference to other software quality improvement techniques, like reviews, inspections and walkthroughs (Myers, 1979). Furthermore, the purpose of testing is to identify failures and problems when the software does not behave as expected. This implies that there is an oracle that knows the expected behavior.

Note that in the context of software testing there are the three different concepts of errors, faults and failures. We use the common notion (IEEE, 1990; Avižienis et al., 2004; Ammann and Offutt, 2008) where a *failure* is the observable incorrect behavior of a program, i.e., an incorrect result. A *fault* is the root cause of the failure, e.g. an incorrect instruction in the source code of a program. The manifestation of a fault, i.e. an incorrect internal software state, is called an *error*.

Having these different concepts allows to clearly distinguish between the activity of testing and the activity of debugging. While testing aims to detect failures, debugging is the process of finding the faults given a set of failures.

2.1 Classifying Software Testing Approaches

There is a great variety of software testing techniques and these techniques apply at different levels of the software development process. A rough, commonly used, classification (Myers, 1979) is to distinguish between *black-box testing*, also known as data-driven or input/output-driven testing, and *white-box testing*. For black-box testing tests from external descriptions are derived, i.e. without looking at the source-code of the program. In contrast to that, for white-box testing tests are derived by considering the software internals. A concept between white-box and black-box testing is gray-box testing Kaner et al. (2001). For *gray-box testing* the test cases are applied from outside of the program, as in black-box testing. However, for the test case selection information from the internals of the program is used.

Another classification refers to the level of abstraction at which the testing activity takes place. The literature (e.g. (Ammann and Offutt, 2008; Utting and Legeard, 2007)) basically distinguishes five different levels which are aligned with the main software development activities:

- *Acceptance Testing* aims to validate if the software system satisfies the user requirements. Does the software do the things, the customer wants it to do?
- *System Testing* is applied to the system as a whole.
- *Integration Testing* is used to check if the various modules and components work together properly. At this level the consistency of the assumptions of the different modules should be validated.
- *Module/Component Testing* tests each module or component separately.
- *Unit Testing* asses the various development units at their finest level. This type of testing can often be conducted without knowing the encapsulating application.

Often software testing is also classified with respect to the tested characteristics, i.e. with respect to the aim of testing. For example, software applications are often assessed regarding the following characteristics (Utting and Legeard, 2007).

- *Usability.* Usability testing has the goal of validating how well the user interface supports the user's needs. For example, how fast can a user perform a particular task using the software's graphical interface.
- *Functional.* Functional testing, also known as behavioral testing, aims to detect incorrect behavior of the system, e.g. where the software produces incorrect outputs.
- *Robustness.* Robustness testing is used to validate the system's behavior under invalid inputs. For example, does the system handle network errors properly?
- *Performance* Performance testing aims to validate the timing requirements of the system. Does the system perform well even under heavy load?

However, there are many other schemes for classifying software testing activities (e.g. (Beizer, 1990)). The aim of this chapter is not to provide an exhaustive list of all possible taxonomies on software testing, but to give enough information for putting the techniques described in this thesis in an appropriate context.

The focus of this thesis is model-based conformance testing of reactive systems. While model-based testing is a black-box testing technique (Utting and Legeard, 2007), it is applicable to any level of abstraction. Model-based testing usually focuses on functional testing, but has also been successfully applied to security testing (e.g. Jürjens (2008)). This thesis concentrates on testing of reactive systems, i.e. testing of systems which are intended to continuously interact with their environment, rather than to produce a final result (Manna and Pnueli, 1995). Such systems are for example implementations of communication protocols or control software for hardware devices.

2.2 Model-based Testing

Figure 2.1 shows a typical model-based testing process, which, according to Utting and Legeard (2007), comprises five steps:

1. *Model* the implementation under test by formalizing the indented behavior.
2. *Generate* test cases from the model under the constraints imposed by the test objectives, e.g, coverage of particular scenarios.
3. *Concretize* the test case such that it can be executed on the implementation. Note that this step is sometimes encoded into a test driver and can be seen as part of the test case execution (Pretschner and Philipps, 2005).
4. *Execution* of the test cases against the implementation under test (IUT).

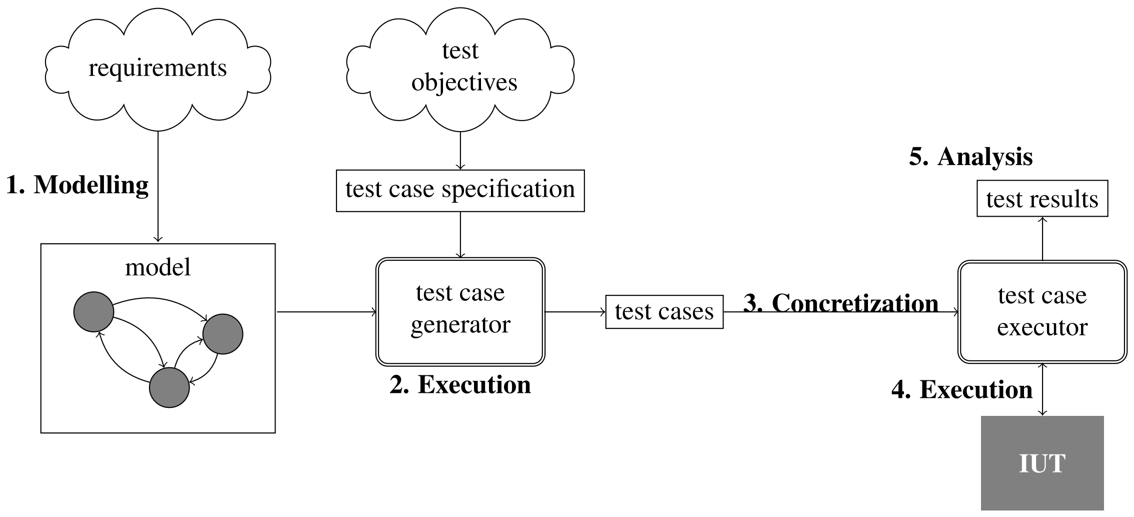


Figure 2.1: A typical model-based testing process. This figure is based on Utting and Legeard (2007) and on Pretschner and Philipps (2005).

5. *Analysis* of the test results for identifing the detected differences between the implementation and its model.

The first step in the model-based testing process is the formalization of the indented behavior of a system in some formal specification language. There is a number of different ways to formalize a system's behavior. One way is to use a state-based description language, like VDM (Jones, 1990) or Z (ISO, 2002) which describe the system in terms of its states and the operations which change the state. While this may lead to infinite state spaces another common modelling paradigm is the use of finite state machines (FSMs) (Lee and Yannakakis, 1996; Hierons et al., 2009). In the presence of concurrency, process algebras (e.g. CSP (Hoare, 1985), CCS (Milner, 1980), LOTOS (ISO, 1989)) are a well suited way for building the formal model. They describe a system in terms of communicating processes. Sometimes systems are modelled solely in terms of algebraic properties, i.e. as a set of axioms (e.g. CASL (Bidoit and Mosses, 2003)).

Once there is a proper model of the system under test, the test case generation should be focused and systematic. Therefore, test case generation is usually guided by some test objectives. One way to guide the test case generation is to use formalized scenarios (e.g. (Jard and Jéron, 2005; Campbell et al., 2005; Grabowski et al., 1993)), i.e., formal specifications of test cases. Another commonly used technique is coverage criteria (Myers, 1979). The derived test cases should cover the important aspect of the specification. Often, one generates test cases that aim to detect particular faults. This is also known as mutation testing and has first been applied to specifications by Budd and Gopal (1985).

Given a formal model and a set of test case specifications there are various techniques for generating test cases. For an overview we refer to state of the art surveys (Hierons et al., 2009; Fraser et al., 2009; Broy et al., 2005).

Despite all these techniques for the generation of test cases based on a formal model, there is still the question of when does a system conform to its specification. While this question is usually answered by the definition of a conformance relation, there are additional issues (e.g. test execution, test verdicts) that need to be addressed in a formal conformance testing framework.

2.3 Formal Conformance Testing

While the ISO (1994) standard 'OSI Conformance Testing Methodology and Framework', which is mainly indented for textual specifications, has been formalized by Tretmans (1992) a framework for formal methods in conformance testing has been proposed in an ISO committee draft (1997) and by Hogrefe et al. (1996).

The central element within this formal conformance testing framework is the definition of what is a correct implementation of a formal specification. This defines a conformance relation. To define such conformance relations it is assumed that there exists a formal specification of the required behavior, i.e., $s \in SPECS$. $SPECS$ is the set of all possible specifications that can be expressed using a particular specification language.

In addition to the specification, there is the implementation under test $iut \in IMPS$, which denotes the real, physical implementation. $IMPS$ is the universe of all possible implementations. Our aim is to formally reason about the correctness of the concrete implementation iut with respect to the specification s. Thus, as a common testing hypothesis (Tretmans, 1992; Bernot, 1991) it is assumed that every implementation can be modeled by a formal object $m_{iut} \in MODS$. $MODS$ denotes the set of all models. Note that it is not assumed that this model is known, only its existence is required.

Conformance is expressed as a relation between formal models of implementations and specifications, i.e.,

$$\mathbf{imp} \subseteq MODS \times SPECS$$

As each model $m_{iut} \in MODS$ represents a concrete implementation $iut \in IMPS$ a conformance relation **imp** allows one to formally reason about the correctness of the iut with respect to a specification $s \in SPECS$.

By applying inputs to the implementation and observing outputs, i.e. by testing, one wants to find non-conforming implementations. The universe of test cases is given by $TESTS$. Executing a test case $t \in TESTS$ on an implementation leads to observations $obs \subseteq OBS$, where OBS denotes the universe of observations.

Formally, test case execution is modeled by a function $exec : TESTS \times MODS \rightarrow OBS$. Given a test case $t \in TESTS$ and a model of an implementation $m \in MODS$, $exec(t,m)$ gives the observations in OBS that result from executing t on the model m.

Finally, there is a function $verd$ that assigns a verdict, i.e. pass or fail, to each observation: $verd : OBS \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$. An implementation $iut \in IMPS$ *passes* a test suite $TS \subseteq TESTS$ if the test execution of all its test cases leads to an observation for which $verd$ evaluates to **pass**. In practice, there is a third verdict, i.e. inconclusive, that is used for judging test executions (Jard and Jéron, 2005). This verdict is used if the implementation has not done anything wrong but the responses of the IUT do not satisfy the test objective.

There are different types of models and conformance relations that can be seen as an instantiation of this formal conformance testing framework. For example, when one uses testing techniques based on finite state machines (FSMs) (Lee and Yannakakis, 1996; Hierons et al., 2008) then $MODS$ and $SPECS$ are some sets of FSMs. Usually, the considered conformance relation is some relation between the states of the implementation and the states of the specification (e.g. isomorphism).

Another instantiation of this formal testing framework uses labeled transition systems for representing specifications and models of implementations. There is a broad range of relations that have been defined for labeled transition systems, e.g. bisimulation equivalence (Milner, 1990), failure equivalence and preorder (Hoare, 1985), and refusal testing (Phillips, 1987), just to name a few.

Chapter 3

Input-Output Conformance Testing

One commonly used conformance relation is the input-output conformance relation (**ioco**) of Tretmans (1996). This relation is designed for functional black box testing of systems with inputs and outputs and is the relation that has been chosen for this thesis.

3.1 Specifications, Implementations and Conformance

The **ioco** relation expresses the conformance of implementations to their specifications where both are represented as labeled transition systems (LTS). Because **ioco** distinguishes between inputs and outputs, the alphabet of an LTS is partitioned into inputs and outputs.

Definition 3.1 (LTS with inputs and outputs) *A labeled transition system with inputs and outputs is a tuple $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, where Q is a countable, non-empty set of states, $L = L_I \cup L_U$ a finite alphabet, partitioned into two disjoint sets, where L_I and L_U are input and output alphabets, respectively. $\tau \notin L$ is an unobservable action, $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ is the transition relation, and $q_0 \in Q$ is the initial state.*

Following the original definitions of Tretmans (1996) we use L_U instead of L_O in order to avoid confusion between L_O and L_0, i.e., between the letter O and the digit zero.

An LTS is *deterministic* if for any sequence of actions starting at the initial state there is at most one successor state.

We use the following common notations:

Definition 3.2 *Given a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ and let $q, q', q_i \in Q, a_{(i)} \in L$ and $\sigma \in L^*$.*

$$\begin{aligned}
q \xrightarrow{a} q' &=_{df} (q, a, q') \in \rightarrow \\
q \xrightarrow{a} &=_{df} \exists q' \bullet (q, a, q') \in \rightarrow \\
q \overset{a}{\not\rightarrow} &=_{df} \nexists q' \bullet (q, a, q') \in \rightarrow \\
q \overset{\varepsilon}{\Longrightarrow} q' &=_{df} (q = q') \vee \exists q_0, \ldots, q_n \bullet (q = q_0 \wedge q_0 \xrightarrow{\tau} q_1 \wedge \cdots \wedge q_{n-1} \xrightarrow{\tau} q_n \wedge q_n = q') \\
q \overset{a}{\Longrightarrow} q' &=_{df} \exists q_1, q_2 \bullet q \overset{\varepsilon}{\Longrightarrow} q_1 \wedge q_1 \xrightarrow{a} q_2 \wedge q_2 \overset{\varepsilon}{\Longrightarrow} q' \\
q \overset{a_1 \ldots a_n}{\Longrightarrow} q' &=_{df} \exists q_0, \ldots, q_n \bullet q = q_0 \wedge q_0 \overset{a_1}{\Longrightarrow} q_1 \wedge \cdots \wedge q_{n-1} \overset{a_n}{\Longrightarrow} q_n \wedge q_n = q' \\
q \overset{\sigma}{\Longrightarrow} &=_{df} \exists q' \bullet q \overset{\sigma}{\Longrightarrow} q'
\end{aligned}$$

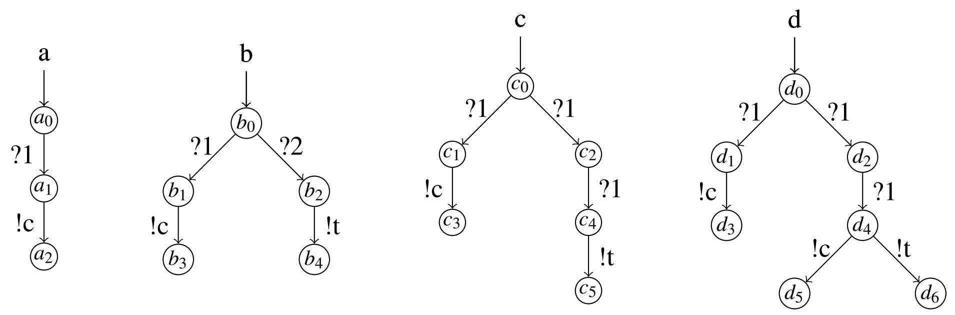


Figure 3.1: Four labeled transition systems.

$init(q)$ denotes the actions enabled in state q. Furthermore, q **after** σ denotes the set of states reachable by a particular trace σ.

Definition 3.3 *Given an LTS $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ and $q, q' \in Q, S \subseteq Q$, $a \in L$, and $\sigma \in L^*$:*

$$\begin{aligned} init(q) &=_{df} \{a | q \xrightarrow{a}\} \\ init(S) &=_{df} \bigcup_{q \in S} init(q) \\ q \textbf{ after } \sigma &=_{df} \{q' | q \overset{\sigma}{\Rightarrow} q'\} \\ S \textbf{ after } \sigma &=_{df} \bigcup_{q \in S} (q \textbf{ after } \sigma) \end{aligned}$$

Note that we will not always distinguish between an LTS M and its initial state and write $M \Rightarrow$ instead of $q_0 \Rightarrow$.

Example 3.1. Figure 3.1 shows four labeled transition systems representing a coffee/tea (c/t) vending machine. The input and output alphabets are given by $L_I = \{1,2\}$ and by $L_U = \{c,t\}$, i.e. one can insert one and two unit coins and the machine outputs coffee or tea. We denote input actions by the prefix "?", while output actions have the prefix "!". For example, a_0 **after** $\langle ?1 \rangle = \{a_1\}$ while c_0 **after** $\langle ?1 \rangle = \{c_1, c_2\}$. □

The **ioco** conformance relation employs the idea of observable quiescence. That is, it is assumed that a special action, i.e. δ, is enabled in the case where the labeled transition system does not provide any output action. These δ-labeled transitions allow to detect implementations that do not provide outputs while the specification requires some output (see Example 4: $\neg(k$ **ioco** $e)$). The input-output conformance relation identifies quiescent states as follows: A state q of a labeled transition system is quiescent if neither an output action nor an internal action (τ) is enabled in q.

Definition 3.4 *Let M be a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, such that $L_I \cap L_U = \emptyset$, then a state $q \in Q$ is quiescent, denoted by $\delta(q)$, if $\forall a \in L_U \cup \{\tau\} \bullet q \overset{a}{\not\rightarrow}$.*

By adding δ-labeled transitions to LTSs the quiescence symbol can be used as any other action.

Definition 3.5 *Let $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ be an LTS then $M_\delta = (Q, L \cup \{\tau, \delta\}, \rightarrow \cup \rightarrow_\delta, q_0)$ where $\rightarrow_\delta =_{df} \{q \xrightarrow{\delta} q | q \in Q \wedge \delta(q)\}$. The suspension traces of M_δ are $Straces(M_\delta) =_{df} \{\sigma \in (L \cup \{\delta\})^* | q_0 \xRightarrow{\sigma}\}$.*

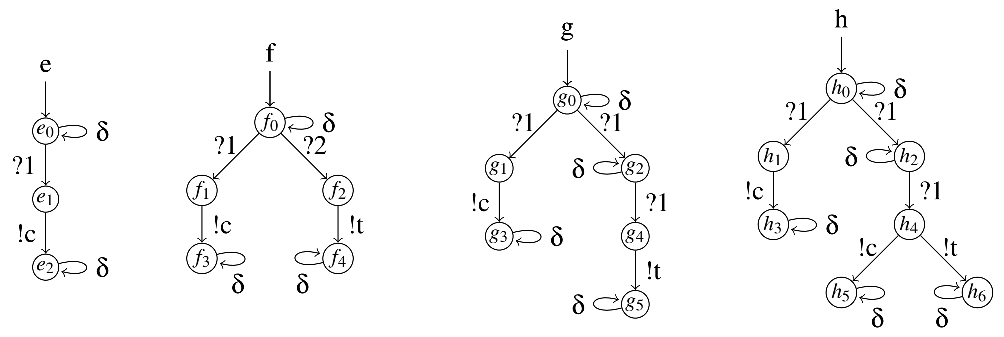


Figure 3.2: δ-annotated labeled transition systems.

Unless otherwise indicated, from now on we include δ in the transition relation of LTSs, i.e., we use M_δ instead of M.

The class of labeled transition systems with inputs L_I and outputs in L_U (and with quiescence) is denoted by $\mathcal{LTS}(L_I, L_U)$ (Tretmans, 1996). In the context of the formal conformance testing framework (see Section 2.3), this is the universe of specifications, i.e. $SPECS = \mathcal{LTS}(L_I, L_U)$.

Example 3.2. Figure 3.2 shows the δ-annotated LTSs for the LTSs illustrated in Figure 3.1. For example, the states g_0, g_2, g_3, and g_5 are quiescent because these states do not have enabled output actions nor enabled τ actions. □

Models for implementations in terms of the input-output conformance relation are input-output transition systems (IOTS). Recall that it is not assumed that these models are known in advance, but only their existence is required. Implementations are not allowed to refuse inputs, i.e. implementations are assumed to be input-enabled and so are their models. Note that specifications do not have to be input-enabled.

Definition 3.6 (Input-output transition system) *An input-output transition system is an LTS $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, such that $L_I \cap L_U = \emptyset$, where all input actions are enabled (possibly preceded by τ-transitions) in all states: $\forall a \in L_I, \forall q \in Q \bullet q \overset{a}{\Longrightarrow}$.*

This sort of input-enabledness, i.e. where τ-labeled transitions may precede input actions ($\forall a \in L_I, \forall q \in Q \bullet q \overset{a}{\Longrightarrow}$), is called *weak input-enabledness*. On the contrary, *strong input-enabledness* requires that all input actions are enabled in all states, i.e. $\forall a \in L_I, \forall q \in Q \bullet q \xrightarrow{a}$.

The class of IOTSs with inputs L_I and outputs in L_U is given by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$ (Tretmans, 1996). As IOTSs are used to formally reason about implementations, input-output transition systems are the implementation models, i.e. $MODS = \mathcal{IOTS}(L_I, L_U)$, when instantiating the formal framework of conformance testing.

Example 3.3. Figure 3.3 depicts the IOTSs derived by making the labeled transition systems of Figure 3.2 weakly input-enabled. □

Before giving the definition of the **ioco** relation we need to define what the outputs of a particular state are and what the outputs of a set of states are; $out(q)$ denotes the outputs at a state q.

Definition 3.7 *Given a labeled transition system $M = (Q, L \cup \{\tau, \delta\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, such that $L_I \cap L_U = \emptyset$, let $q \in Q$ and $S \subseteq Q$, then*

$$\begin{aligned} out(q) &=_{df} \{a \in L_U \mid q \xrightarrow{a}\} \cup \{\delta \mid \delta(q)\} \\ out(S) &=_{df} \bigcup_{q \in S} (out(q)) \end{aligned}$$

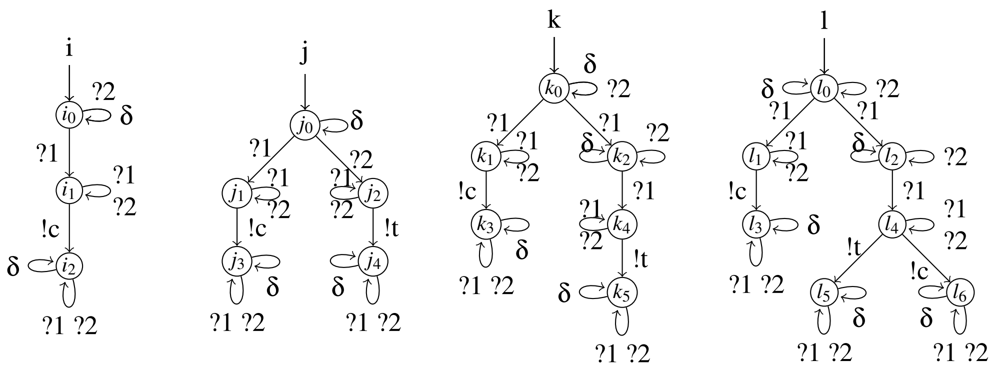

i
j
k
l
i_0 i_1 i_2 j_0 j_1 j_2 j_3 j_4 k_0 k_1 k_2 k_3 k_4 k_5 l_0 l_1 l_2 l_3 l_4 l_5 l_6
?1 ?2 !c !t δ

Figure 3.3: Input-output transition systems.

Informally, the input-output conformance relation states that an implementation under test (IUT) conforms to a specification S iff the outputs of the IUT are outputs of S after an arbitrary suspension trace of S. Formally, **ioco** is defined as follows:

Definition 3.8 (Input-output conformance) *Given a set of inputs L_I and a set of outputs L_U then* **ioco** $\subseteq$ $\mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ *is defined as:*

$$IUT \ \textbf{ioco} \ S =_{df} \forall \sigma \in Straces(S) \bullet out(IUT \ \textbf{after} \ \sigma) \subseteq out(S \ \textbf{after} \ \sigma)$$

Example 3.4. Consider the LTSs of Figure 3.2 to be specifications and let the IOTSs of Figure 3.3 be implementations. Then we have i **ioco** e and j **ioco** f. We also have j **ioco** e because $\langle ?2 \rangle$ is not a trace of e. Thus, this branch is not relevant with respect to **ioco**. k does not conform to e, i.e. $\neg(k \ \textbf{ioco} \ e)$, because $out(k \ \textbf{after} \ ?1) = \{!c, \delta\} \not\subseteq \{!c\} = out(e \ \textbf{after} \ ?1)$. Furthermore $\neg(l \ \textbf{ioco} \ e)$ because $out(l \ \textbf{after} \ ?1) = \{!c, \delta\} \not\subseteq \{!c\} = out(e \ \textbf{after} \ ?1)$. Due to the use of suspension traces we also have $\neg(l \ \textbf{ioco} \ k)$ because $out(l \ \textbf{after} \ \langle ?1, \delta, ?1 \rangle) = \{!c, !t\} \not\subseteq \{!t\} = out(k \ \textbf{after} \ \langle ?1, \delta, ?1 \rangle)$. □

The **ioco** definition from above can be lifted to a more general definition where different instantiations correspond to different conformance relations:

Definition 3.9 (Generic input-output conformance) *Given a set of inputs L_I and a set of outputs L_U then* $\textbf{ioco}_{\mathcal{F}} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ *is defined as:*

$$IUT \ \textbf{ioco}_{\mathcal{F}} \ S =_{df} \forall \sigma \in \mathcal{F} \bullet out(IUT \ \textbf{after} \ \sigma) \subseteq out(S \ \textbf{after} \ \sigma)$$

Using $\textbf{ioco}_{\mathcal{F}}$ we can now express different relations by selecting a proper set of sequences for $\mathcal{F}$. The input output testing relation ($\leq_{iot}$) is given by $\textbf{ioco}_{A^*}$, while the input output refusal relation ($\leq_{ior}$) can be defined as $\textbf{ioco}_{(A \cup \{\delta\})^*}$. **ioconf** is given by $\textbf{ioco}_{traces(S)}$.

Example 3.5. The input-output transition systems k and l of Figure 3.3 serve to illustrate the difference between $\leq_{ior}$, $\leq_{iot}$, **ioco**, and **ioconf**. Under all of these conformance relations k conforms to l, thus we have, $k \leq_{iot} l$, $k \leq_{ior} l$, k **ioconf** l, and k **ioco** l. In difference to that, we have $l \leq_{iot} k$ but $\neg(l \leq_{ior} k)$ because $out(l \ \textbf{after} \ \langle ?1, \delta, ?1 \rangle) = \{!c, !t\} \not\subseteq \{!t\} = out(k \ \textbf{after} \ \langle ?1, \delta, ?1 \rangle)$. l conforms to k with respect to **ioconf**, i.e. l **ioconf** k. Note that l is not conform to k regarding to **ioco**, i.e., $\neg(l \ \textbf{ioco} \ k)$, because $out(l \ \textbf{after} \ \langle ?1, \delta, ?1 \rangle) = \{!c, !t\} \not\subseteq \{!t\} = out(k \ \textbf{after} \ \langle ?1, \delta, ?1 \rangle)$. Furthermore, **ioconf** and **ioco** allow the use of incomplete specifications. For example, we have j **ioconf** i and j **ioco** i but $\neg(j \leq_{iot} i)$ and $\neg(j \leq_{ior} i)$. This is because the conformance relations **ioconf** and **ioco** only consider (suspension) traces of the specification while $\leq_{iot}$ and $\leq_{ior}$ consider all possible (suspension) traces.

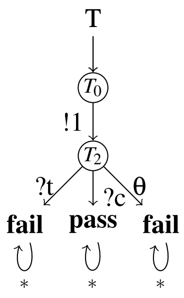


Figure 3.4: Example of a test case.

3.1.1 Test Cases and Test Case Execution

By using a particular set of test cases one wants to test if a given implementation conforms to its specification. In the **ioco** framework a test case is also a labeled transition system (Tretmans, 1996). In a test case the observation of δ is implemented by θ, i.e., test cases use θ, to observe δ. This is because, in practice δ is a timeout, which is not a normal event that can be observed. θ can be seen as the timer used to observe the occurrence of quiescence, i.e. the occurrence of δ. Note that inputs of a test case are outputs of an IUT and vice versa.

Definition 3.10 (Test case) *A test case T is an LTS $T = (Q, L \cup \{\theta\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, and $L_I \cap L_U = \emptyset$, such that (1) T is deterministic and has finite behavior; (2) Q contains sink states **pass** and **fail** (in **pass** and **fail** states a test case synchronizes on any action); and (3) for any state $q \in Q$ where $q \neq$ **pass** and $q \neq$ **fail**, either $init(q) = \{a\}$ for some $a \in L_U$, or $init(q) = L_I \cup \{\theta\}$.*

Jard and Jéron (2005) call property (3) of this definition *controllability*, i.e. a test case is controllable if it does not contain choices between inputs and outputs.

$\mathcal{TEST}(L_U, L_I)$ denotes the class of test cases over L_U and L_I, i.e. $TESTS = \mathcal{TEST}(L_U, L_I)$. A *test suite* is a set of test cases.

Example 3.6. Figure 3.4 shows a test case satisfying Definition 3.10. In a state either inputs (outputs of the IUT) and the θ event are enabled, or an output is enabled. The $*$-labeled transitions in **pass** and **fail** states denote that in such states a test case synchronizes on any action. □

Running a test case $t \in \mathcal{TEST}(L_U, L_I)$ against an implementation under test $i \in \mathcal{IOTS}(L_I, L_U)$ is similar to the parallel composition of CSP (cf. (Hoare, 1985)) of the test case and the implementation. The only difference is that θ is used to observe δ. Formally, running t on i is denoted by $t \rceil\!| i$.

Definition 3.11 (Synchronous execution) *Given a test case $t \in \mathcal{TEST}(L_U, L_I)$, an IUT $i \in \mathcal{IOTS}(L_I, L_U)$, and let $a \in L_U \cup L_I$, then the synchronous test case execution operator $\rceil\!|$ has the operational semantics defined by the following inference rules:*

$$\frac{i \xrightarrow{\tau} i'}{t \rceil\!| i \xrightarrow{\tau} t \rceil\!| i'} \qquad \frac{t \xrightarrow{a} t',\ i \xrightarrow{a} i'}{t \rceil\!| i \xrightarrow{a} t' \rceil\!| i'} \qquad \frac{t \xrightarrow{\theta} t',\ i \xrightarrow{\delta}}{t \rceil\!| i \xrightarrow{\theta} t' \rceil\!| i}$$

A test run can always continue, i.e. there are no deadlocks. This is because a test case synchronizes on any action when a verdict state is reached. Formally, a test run is a trace of $t \rceil\!| i$ leading to a verdict state (**pass**, **fail**) of t:

Definition 3.12 (Test run) *Given a test case $t \in \mathcal{TEST}(L_U, L_I)$ and an IUT $i \in \mathcal{IOTS}(L_I, L_U)$, then a test run $\sigma \in L_U \cup L_I \cup \{\theta\}$ is given by: $\exists i' \bullet t \rceil | i \overset{\sigma}{\Longrightarrow} \textbf{pass} \rceil | i'$ or $\exists i' \bullet t \rceil | i \overset{\sigma}{\Longrightarrow} \textbf{fail} \rceil | i'$.*

An implementation *i* passes a test case iff every possible test run leads to a **pass** verdict state of the test case:

Definition 3.13 (Passing) *Given an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ and a test case $t \in \mathcal{TEST}(L_U, L_I)$, then*

$$i\ \textbf{passes}\ t \Leftrightarrow_{df} \forall \sigma \in (L_I \cup L_U \cup \theta)^*, \forall i' \bullet t \rceil | i \overset{\sigma}{\not\Longrightarrow} \textbf{fail} \rceil | i'$$

Example 3.7. Running the test case of Figure 3.4 on the IUT *i* of Figure 3.3 leads to the following test run:

$$T_0 \rceil | i_0 \overset{\langle !1, ?c \rangle}{\Longrightarrow} \textbf{pass} \rceil | i_2$$

Because this is the only possible test run and it leads to pass, we have *i* **passes** *T*. □

3.2 Variations of IOCO

Although the **ioco** testing theory is applicable to many types of systems and has been successfully applied in many industrial case studies (e.g. (Philipps et al., 2003; Fernandez et al., 1997; Kahlouche et al., 1999; Prenninger et al., 2005)), it abstracts from various aspects. For example, **ioco** does not consider time, i.e. one cannot use **ioco** for testing constraints like there has to be an output within five seconds. Furthermore, input-output conformance testing is not intended for distributed environments, where a test case cannot directly access all points of control and observation of the implementation under test.

Consequently, various extensions to input-output conformance testing have been made. Some of them extend and modify the ioco relation, while others only change the underlying type of transition system. Note that there are many more preorder relations that are suited for testing. It is out of the scope of this thesis to review all these relations; for an overview of other relations we refer to Bruda (2005).

3.2.1 Symbolic Input-Output Conformance (sioco)

Labeled transition systems tend to be rather huge for many specifications since they explicitly contain every possible transition. In other words, using data within an LTS results in one edge for every valuation of the used variables, since the framework of labeled transition systems does not provide a concrete notion of data and data-dependent control flow.

Due to this explicit representation of data, industrial specifications typically suffer from the state space explosion problem. The idea of using symbolical representations for data within the transition system has been considered by Frantzen et al. (2004) and by Rusu et al. (2000). The used models, i.e. symbolic transition systems, are transition systems with an explicit notion of data and data-dependent control flow.

Example 3.8. Figure 3.5 shows an example symbolic transition system, representing a drink vending machine. Transitions are labeled with an action. Guard conditions are included in brackets [] and followed by the update function. Within the illustrated transition system the variable *v* stores the number of inserted coins. For example, when coins are inserted into the machine (self-loop transition on state m_0) *v* is increased. The guard of this transition states that at most two coins can be inserted. □

The semantics of a symbolic transition system can be defined in terms of a labeled transition system. Frantzen et al. (2004) define the interpretation of a symbolic transition system $\mathcal{S}$ as an LTS $[\![\mathcal{S}]\!]$, which is derived by evaluating all variables and unfolding all symbolic states.

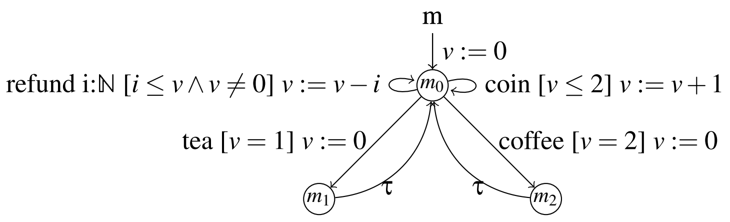

m
$v := 0$
refund i:$\mathbb{N}$ $[i \leq v \wedge v \neq 0]$ $v := v - i$
m_0
coin $[v \leq 2]$ $v := v + 1$
tea $[v = 1]$ $v := 0$
coffee $[v = 2]$ $v := 0$
τ
τ
m_1
m_2

Figure 3.5: Example of a symbolic transition system representing a vending machine.

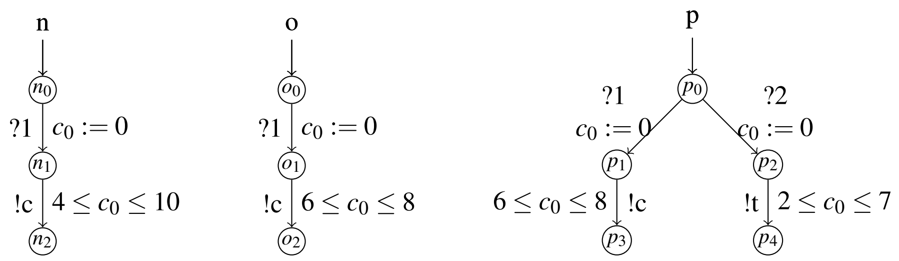


Figure 3.6: Examples of three timed automata with inputs and outputs.

Frantzen et al. (2004) initially defined the conformance of an implementation with respect to a symbolic transition system by means of **ioco** and the interpretation of the STS. Later Frantzen et al. (2006) introduced the conformance relation **sioco** which directly relates symbolic transition systems. However, **sioco** coincides with **ioco**. So informally, **sioco** is defined as follows:

Definition 3.14 *An implementation under test (IUT) conforms to a specification S iff the outputs of the IUT are outputs of S after an arbitrary symbolic suspension trace of S.*

3.2.2 Timed Input-Output Conformance

Recently, testing under the presence of time has received much attention (e.g. (Krichen and Tripakis, 2004; Briones and Brinksma, 2004; Khoumsi et al., 2004; Hessel et al., 2008)).

Basically, there are two different frameworks that haven been proposed for conformance testing under the presence of real-time constraints. While the framework for testing real time systems of Briones and Brinksma (2004) is based on timed labeled transition systems, Krichen and Tripakis (2004) rely on timed automata with deadlines of Alur and Dill (1994). The framework based on timed automata does not consider the absence of outputs, i.e. quiescence. On the contrary, the framework of Briones and Brinksma (2004) considers quiescence, but has no notion of deadlines. However, the main ideas behind these two frameworks are similar.

Example 3.9. Figure 3.6 shows three different timed automata with inputs and outputs. For the sake of simplicity we omit timing constraints on these automata. The first automata describes a system where after the insertion of a coin (?1) the system must output coffee (!c) no earlier than four and no later than ten time units. □

A timed automaton represents a set of real-time sequences, which comprise observable events and the passage of time. Informally, timed input-output conformance between an implementation and a specification is given as follows.

Definition 3.15 (tioco) *An implementation under test (IUT) conforms to a specification S iff the observable outputs of IUT, i.e., output actions or the passage of time, are allowed by the specification after any trace of S.*

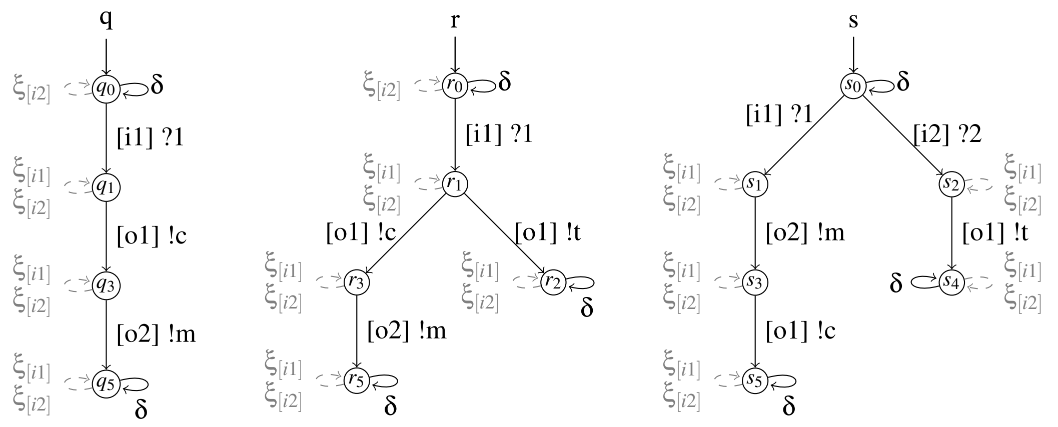


Figure 3.7: Examples of multi input-output transition systems representing a vending machine.

Example 3.10. Let n be the specification and o and p be implementations. Note that **tioco** requires implementations to be input-complete; for the sake of simplicity our figures do not show the additional edges caused by this requirement. We have o **tioco** n because $!c$ comes not before 4 time units but certainly before 10, because it is issued between 6 and 8 time units. We also have p **tioco** n because the trace starting with ?2 is not a trace of a and thus not relevant. If p is the specification then neither n nor o is conform (with respect to **tioco**) because both do not react on the insertion of a 2 coin. □

3.2.3 Distributed and Multi-Port Input-Output Conformance

Testing is sometimes conducted within a distributed system where the points of control and observation (PCO) are distributed over geographically distinct places. There are different test architectures that deal with such systems. A local test architecture assumes that, although there are multiple points of control and observation, a single tester has access to these PCOs. Testing under this architecture has been investigated by Heerink and Tretmans (1997) and leads to the conformance relation **mioco**. In a distributed test architecture there are multiple testers which are connected to the different PCOs. Hierons et al. (2008) presented a conformance relation, i.e. **dioco**, that is suitable for testing in a distributed test architecture.

Regardless of the used test architecture, the formal models are multi input-output transition systems which are in fact input-output transition systems with partitioned input and output alphabets. While Hierons et al. (2008) assign each PCO one input partition and one output partition, Heerink and Tretmans (1997) consider each partition as distinct channel.

Example 3.11. Figure 3.7 shows three multi-port input-output labeled transition (MIOTS) systems. The MIOTS use four different ports: [i1] and [i2] are input ports, e.g. two different money slots of the drink vending machine; [o1] and [o2] are the two output ports. For example the MIOTS s takes either a one coin at port [i1] or a two coin at port [i2]. If a user inserts a one coin, then the machine delivers first coffee at the coffee/tea outlet ([o1]) and then milk at the outlet [o2]. The ξ events denote input suspension, which is needed for the **mioco** conformance relation. □

The **mioco** conformance relation of Heerink and Tretmans (1997) considers refusal testing (Phillips, 1987) of multi-port input-output transition systems. Traces are given in terms of failure traces (Baeten and Weijland, 1990). A failure trace consists of actions and sets of refused actions. The multi-port input-output transition systems used by Heerink and Tretmans (1997) are strongly input-enabled for channels and states where inputs are possible. In addition to quiescence, i.e. output suspension, they introduce a notion of input suspension, i.e. the inability of a particular channel to accept an input.

Definition 3.16 (mioco) *An implementation under test (IUT) conforms to a specification S iff the outputs of the implementation, i.e. output events, output suspension and input suspension, are allowed by the specification for any failure trace of the specification.*

Example 3.12. The multi-port input-output transition systems of Figure 3.7 serve to illustrate the **mioco** relation. For the sake of simplicity, we omit the labels required due to input-enabledness in our multi-port input-output transition systems. We have, q **mioco** r. r is not a valid implementation of q, because after inserting a one coin at [i1] one gets tea (!t) or coffee (!c) but the specification q only allows for coffee (!c). Neither q nor r are valid implementations of s with respect to **mioco**, i.e. $\neg(q$ **mioco** $s)$ and $\neg(r$ **mioco** $s)$, because from q and r one can observe input suspension on port [i2] (i.e. $\xi_{[i2]}$) at state q_0 and r_0, but this is not allowed at state s_0. □

Heerink and Tretmans (1997) present a simple test case generation algorithm for **mioco** based test cases. The efficiency of this approach has been improved by Brinksma et al. (1998) and by Li et al. (2004).

The conformance relation **mioco** assumes that all output channels of the implementations are lockable, i.e. the environment can block the channel such that the channel will not produce any output. Li et al. (2003) presented an extension to **mioco** by allowing non-lockable output channels. A non-lockable output channel is a channel that cannot be blocked by the environment.

Test cases generated with respect to **mioco** are not distributed, i.e. they apply to local test architectures only. The distribution of test cases derived for **mioco** conformance testing has been studied in Li et al. (2004). The authors of this work present a distribution rule that allows the distribution of a centralized test case. Their approach relies on a hand written election service, which is responsible for selecting the currently active test case, and requires a controllable centralized test case.

Recently, testing multi-port transition systems under the presence of time has been considered by Briones and Brinksma (2005).

Hierons et al. (2008) adopt the **ioco** relation to the distributed test architecture. Their **dioco** relation considers controllability and observability problems (Cacciari and Rafiq, 1999) which occur when testing in distributed environments.

Definition 3.17 (dioco) *An implementation under test (IUT) conforms to a specification S if for every suspension trace σ of the IUT ending in a quiescent state, if there is a suspension trace comprising the same order of input events (regardless of possibly interleaved output events) in the specification, then the specification has a suspension trace, which projection to the ports is equivalent to the port projection of σ.*

Example 3.13. Again we use the multi-port input-output transition systems of Figure 3.7 to illustrate conformance with respect to **dioco**. Note that **dioco** does not consider input suspension. Thus, all gray colored edges are not relevant for this conformance relation.

We have q **dioco** r. r does not conform to q, i.e. $\neg(r$ **dioco** $q)$, because for the trace $\langle [i1]?1, [o1]!t \rangle$ of r there is a trace in q with the same input events (e.g $\langle [i1]?1 \rangle$) but there is no trace in q that has the same projections on the ports as $\langle [i1]?1, [o1]!t \rangle$.

Interestingly, we have s **dioco** q because for any trace starting with $\langle [i2]?2 \rangle$ there is no trace with the same order of inputs in q, i.e. this traces do not affect conformance. For the trace $\langle [i1]?1, [o2]!m, [o1]!c \rangle$ there is a trace with the same order of inputs in q, i.e. $\langle [i1]?1 \rangle$. Furthermore, the trace $\langle [i1]?1, [o1]!c, [o2]!m \rangle$ of q has the same port projections as the trace of s, i.e. $\langle [i1]?1 \rangle$ for [i1], $\langle [o1]!c \rangle$ for [o1], and $\langle [o2]!m \rangle$ for [o2]. This illustrates one important property of **dioco**: the order of interleaved outputs on different ports does not matter.

Nevertheless, q does not conform to s and r does not conform to s with respect to **dioco**, i.e. $\neg(q$ **dioco** $s)$ and $\neg(r$ **dioco** $s)$. □

The test case generation approach of Hierons et al. (2008) leads to a set of distributed test cases, one test case for each point of control and observation. The test cases are not synchronized. On an correct (distributed) implementation, all test cases will end in pass verdict state.

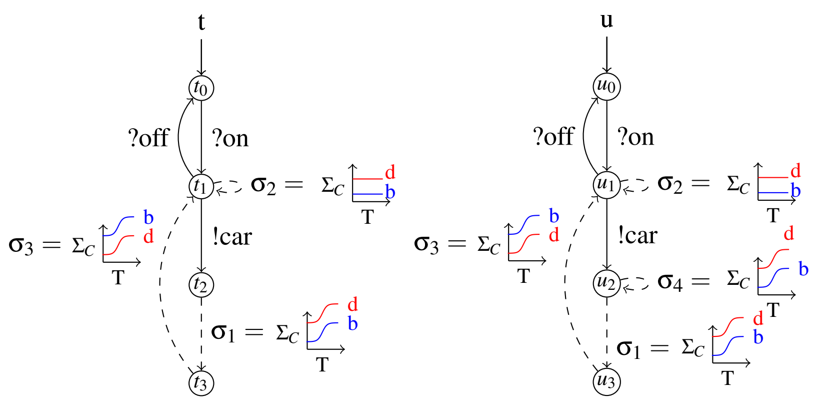


Figure 3.8: Examples of hybrid transition systems.

3.2.4 Hybrid Input-Output Conformance

While labeled transition systems deal with testing of discrete event systems they are not suitable for specifying continuous behavior. However, many systems do not only deal with discrete events but also with continuous inputs and outputs. Hybrid transition systems aim to represent systems that combine both, discrete and continuous behavior. Such systems arise when discrete control software is combined with the interaction of mechanical, electrical or chemical components.

One such example is a break control system of a car (van Osch, 2006), which takes care of a proper distance between the controlled car and any leading car. Discrete events may be ”ON”, ”OFF”, and a ”leading car dectected” event. In contrast to these events, the system receives a continuous signal from the ”distance sensor” and continuously controls the break pressure.

Cuijpers et al. (2002) propose hybrid transition systems for the modelling of systems with discrete and continuous domains.

Example 3.14. Figure 3.8 shows four different hybrid transition systems (HTSs). Discrete actions are denoted by solid lines while dashed lines are used for continuous behavior. The HTS t is a specification for a break control system of a car. The system may be turned on and off (discrete inputs). If a leading car is detected and the system had been turned on, an leading car detected event is issued (!*car*, discrete output). If the system is on and no leading car has been detected (state t_1), then for a default value of the distance sensor (red lines (d); continuous input) the break pressure (blue lines (b)) is (almost) zero. In state t_2 the distance to a leading car may decrease. In that case the system has to increase the break pressure. The HTS u is almost the same as the HTS t. However, in the state u_2 it may choose between increasing the break pressure if the distance to the leading care decreases or keeping the break pressure on the same level. □

A conformance relation for hybrid transition systems (**hioco**) has been presented by van Osch (2006).

Definition 3.18 (hioco) *Informally, an implementation under test (IUT) is* ***hioco****-conform to a specification if the outputs of the implementation are a subset of the outputs of the specification after a particular trace of the specification. In addition, the trajectories of the implementation, filtered by allowed inputs on the specification, must be allowed by the specification as well.*

Note that some of the underlying assumptions of the **hioco** relation make the hybrid transition system infinite.

Example 3.15. We use the hybrid transition systems of Figure 3.8 to illustrate conformance with respect to **hioco**. Note that the hybrid transition system do not satisfy all required properties for **hioco**, because this would render them infinite.

t conforms to u with respect to **hioco**, i.e. t **hioco** u. However, the hybrid transition system u does not conform to t, i.e. $\neg(u$ **hioco** $t)$ because the implementation has σ_1 (increases the break pressure when the distance decreases) and σ_4 (do not change the break when the front car comes closer) enabled in state u_2. As the distance value is an allowed input on the specification, these two behaviors have to be allowed by t. However, t does not allow to keep the break pressure at the same level. □

3.3 Issues in Conformance Testing

Despite all formal conformance relations there are many issues that arise when applying formal conformance testing in practice. Having a precise definition of conformance, i.e. when is an implementation correct with respect to a particular specification, is important but not sufficient for industry. Often the underlying assumptions and their implications of conformance relations are too strong. Another issue is the problem of test case selection. Which test cases should be used and how can they be obtained efficiently? This section reviews identified problems and proposed solutions.

3.3.1 Testing Assumptions

Gaudel (1995) showed that testing assumptions directly relate to the possibility of detecting errors. Different assumptions imply different constraints on the generated test cases. The input-output conformance relation has two main assumptions that are often considered as impractical in the literature (Petrenko and Yevtushenko, 2002; Segala, 1997; Lestiennes and Gaudel, 2005): input-enabledness and synchronous test case execution.

Input-Enabledness

Implementations, and thus also their models *MODS* are required to be input-enabled. That is, in any state an implementation has to accept any input. This assumption simplifies testing because a tester never needs to take care of the allowed inputs.

Different approaches have been proposed to overcome the limitations of input-enabledness. Huo and Petrenko (2004, 2005, 2009) consider blocked and unspecified input actions. They distinguish between the two different types of missing input actions as follows: If no input-actions are enabled in a particular state, then this state blocks input actions. If there is an input action enabled in a particular state, then all other (not explicitly enabled) input actions are unspecified. An implementation may behave arbitrarily, after an unspecified input action. The presented approaches do not explicitly test for the absence of input actions.

Testing non-input-enabled systems, including testing for the refusal of inputs, has been considered by Lestiennes and Gaudel (2005). They introduce a variant of input-output labeled transition systems which allow to specify accepted inputs as well as impossible ones. Impossible inputs are either inputs that are accepted by the implementation but have no effect, or inputs that are not accepted at all, i.e. refused inputs. Their work is a combination of testing with inputs and outputs (Tretmans, 1996) and refusal testing (Phillips, 1987).

Bourdonov et al. (2006) make a different distinction between impossible inputs. They distinguish between forbidden, refused and erroneous inputs. Forbidden inputs are events that are not allowed due to various reasons, i.e. such an input may destroy the implementation. Refused inputs are, similar to Lestiennes and Gaudel (2005) inputs that are not accepted by the implementation. However, only if an implementation accepts an erroneous inputs, the implementation should be considered as incorrect. Bourdonov et al. (2006) define a conformance relation that considers these different types of inputs and show how to generate test cases for this new relation.

Synchronous vs. Asynchronous Environments

Synchronous communication requires that communication between a test case and the implementation under test (IUT) occurs only if both processes offer to interact on a particular action. If an interaction takes place, it happens simultaneously in both, the test case and the IUT. In practice this is often not the case, e.g. when communication is implemented by the use of TCP or UDP.

Test cases are not allowed to comprise choices between inputs and outputs. A tester should always be able to decide deterministically what to do. As identified by Petrenko et al. (2003), this structure of test cases together with input-enabledness leads to test cases that require to prevent an implementation from doing outputs. Thus, if an implementation has a choice between inputs and outputs the test case controls that choice. This issue is especially challenging when the IUT and the tester communicate via TCP (or UDP).

Based on the findings of Petrenko et al. (2003) test cases have been made input-enabled (Tretmans, 2008). Input-enabled test cases do not block outputs of the implementation anymore. As a consequence test cases comprise non-deterministic choices between inputs and outputs and asynchronous communication may lead to incorrect verdicts, because a tester may opt for the wrong transition.

Tan and Petrenko (1998) propose to simply terminate testing with an inconclusive verdict when the system produces an output instead of consuming the input.

Petrenko and Yevtushenko (2002); Petrenko et al. (2003) use two separate test processes for testing. One test process provides inputs to the IUT while the second test process collects the outputs from the IUT. Both test processes communicate with the IUT via finite queues. However, the implementation relation that can be checked by separated test processes is coarser than **ioco**.

Verhaard et al. (1993) have shown that the implementation relations used for synchronous testing are not testable in an asynchronous environment in general. Generated test cases may reject correct implementations when executed in asynchronous environments. In contrast to that, Jard et al. (1999) showed that asynchronous testing can be as powerful as remote testing if a global stamping mechanism is used. That is, the messages are marked with a logical counter at the side of the implementation. The tester then reconstructs the real ordering of the occurred events. Thus, by the use of this approach test cases designed for a synchronous environment can be applied in an asynchronous environment.

Huo and Petrenko (2009, 2005, 2004) rely on input-progressive labeled transition systems, i.e. LTSs which eventually consume their inputs. Basically, the suspension traces of labeled transition systems are related to such traces of the LTS in queued context. While Huo and Petrenko (2005) considers only full specified labeled transition systems, partial specifications are addressed by Huo and Petrenko (2009). The considered conformance relation is queue-context quiescent trace inclusion, which is weaker than the input-output conformance relation.

3.3.2 Distributed Testing

An issue closely related to synchronous and asynchronous test case execution is the issue of distributed testing. While this has been addressed by multi-port input-output transition systems and conformance relations on these systems (see Section 3.2.3), the distribution of centralized test cases has been an active area of research.

Jard et al. (1998) presented an approach for distributing a centralized test case for the **ioconf** relation. The idea is to generate trace equivalent distributed test cases which rely on a hand written election service that is responsible for the coordination of the distributed test cases.

Tanguy et al. (2000) propose a distribution algorithm that integrates the coordination into the different test cases automatically. The synthesized test cases do not rely on an external coordination procedure. Instead, the test cases use synchronization messages and local timers for the coordination among them.

However, the order in which the control has to be passed successively to each local tester, has to be specified manually. A case study of this approach has been presented in Viho (2005).

Both approaches (Jard et al., 1998; Tanguy et al., 2000) distributed test cases for testing input-output conformance with respect to the **ioconf** relation.

A technique that does not distribute a centralized test case, but generates distributed test cases directly from a specification has been proposed by Jard (2003). This approach relies on true concurrency models (Ulrich and König, 1997) for test case generation. At an intermediate level this approach comes up with an abstracted event structure, which contains all the causal relation of the different events. This event structure is then projected on the different testers in order to obtain distributed test cases.

Haar et al. (2007) introduces input-output partial order automata which capture the nature of a distributed environment. Inputs my arrive asynchronously causing the automata to partially follow the transitions. Outputs are sent as soon as they occur on a transition. However, inputs on a particular channel still precede the outputs of this channel.

3.3.3 Compositional Testing and Action Refinement

Another interesting issue is compositional testing: Can we test the components of a software system separately and then conclude that a system composed of these components is conform with respect to the composed specifications?

For **ioco** this question has been investigated by van der Bijl et al. (2003): **ioco** is compositional if both, specifications and implementations are input-enabled. Furthermore, van der Bijl et al. (2003) propose a completion operator that turns a (not input-enabled) labeled transition system into an (input-enabled) input-output transition system. While this operator preserves conformance it makes some non-conforming implementations conform to the specification.

Bourdonov et al. (2006) define two completion operators which preserve conformance as well as non-conformance. While both completion operators first add additional transitions that do not change the trace semantics they then proceed differently. The first operator adds every possible behavior to every unspecified action. The second operator adds for every unspecified input a transition to a special state that has an error-labeled self-loop. By these transformations the authors preserve conformance and non-conformance of completed labeled transition systems.

While compositional testing is a bottom-up approach, i.e. conform components lead to a conform system, refinement is a top-down approach. One starts with an abstract specification and refines the specification until the actions described by the specification are concrete enough to extract test cases that can be applied to an implementation. However, rewriting the model by hand is a time-consuming and error-prone task.

Van der Bijl et al. (2005) propose an approach to automatically obtain test cases at the required level of detail from an abstract model. One input action is refined by using a linear trace of other input actions. van der Bijl et al. (2005) investigate the effect of such refinements on conformance and introduce a conformance relation that uses the refinement information for relating an abstract specification and a concrete implementation.

Action refinement for symbolic transition systems has been considered by Faivre et al. (2008).

3.3.4 Test Case Selection

An important issue when applying conformance testing in industry is the selection of a proper set of test cases. In the context of conformance testing there is a large body of literature dealing with different test case selection techniques. In this section we survey the work of test purpose based, coverage based and fault-based test case selection.

Test Purpose based

Test purposes have been presented as a solution to avoid the state space explosion when selecting test cases from formal models. A test purpose can be seen as a specification of a test case. In conformance testing the notion of test purposes has been standardized (ISO, 1994):

Definition 3.19 (Test purpose, informal) *A description of a precise goal of the test case, in terms of exercising a particular execution path or verifying the compliance with a specific requirement.*

De Vries and Tretmans (2001) extend the conformance testing framework of Tretmans (1999) by the general concept of observation objectives. An observation objective describes what a tester likes to observe from an implementation. Observation objectives are related to test purposes and can be used for test case selection. By integrating observation objectives into the conformance testing framework one can formally reason about testing with respect to observation objectives. The work of Ledru et al. (2001) formally explores the relation between test cases, test purposes and specifications.

Tools like TGV (Jard and Jéron, 2005), SAMSTAG (Grabowski et al., 1993), Microsoft's SPECEXPLORER (Grieskamp et al., 2005; Campbell et al., 2005), STG (Clarke et al., 2002), and others rely on formal representations of test purposes in order to restrict the models to manageable junks.

Various representations of test purposes have been proposed. Test purposes are often formulated on the semantical level of the specification, e.g. test purposes are labeled transition systems (Jard and Jéron, 2005), or symbolic transition systems (Clarke et al., 2002; Rusu et al., 2000; Jeannet et al., 2005). But also other representations have been used. However, high-level representations of test purposes are usually transformed to some lower level representation, e.g. to labeled transition systems. For example, Amyot et al. (2005) derive test purposes from Use Case Maps (Buhr and Casselman, 1996). Da Silva and Machado (2006) presented an approach allowing to transform CTL (Computational Tree Logic, (Clarke et al., 1986)) formulas to test purposes. Grabowski et al. (1993) represent test purposes as message sequence charts (MSC, (ITU-T, 1998)).

The SPECEXPLORER tool does not rely on a single representation of a test purpose but combines various techniques for controlling the test case selection (Campbell et al., 2005). For example, they use state filtering, i.e. they prune away states that do not satisfy a user specified criterion. They also allow the user to select a representative, finite set of data values for variables.

While the TGV tool of Jard and Jéron (2005) relies on labeled transition systems test purpose based test case selection based on symbolic specifications has been considered in (Rusu et al., 2000; Jeannet et al., 2005; Gaston et al., 2006).

Rusu et al. (2000) use symbolic test purposes on symbolic specifications. By calculating the synchronous product between a given test purpose and a specification they obtain the symbolic traces selected by the test purpose from the specification. This synchronous product is then simplified in order to obtain a valid test case: Internal actions and non-deterministic choices get eliminated. The final test case is obtained by selecting parts that lead to the *Accept* locations of the test purpose. During this selection the test case is made input-complete, i.e. missing transitions are added such that they lead to a new location *fail*. Although, the selected test cases satisfy all necessary properties (e.g. they are initialized, they are deterministic and they are input-complete), they may comprise unreachable parts, redundant variables, and complex guards which can be simplified. Rusu et al. (2000) report on first experiments using the HyTech model checker (Henzinger et al., 1997) and the PVS theorem prover (Owre et al., 1992) for simplifying test cases of their case study.

Jeannet et al. (2005) extend the approach of test purpose based test selection to infinite-state symbolic models. In that case the selection of states leading to *Accept*, i.e. the selection of coreachable states, becomes undecidable. Thus, the coreachability analysis is approximated by the use of Abstract Interpretation techniques (Cousot and Cousot, 1977). The set of coreachable states serves as basis for test case selection and for assigning Inconclusive verdicts. Finally, test cases are made input-complete.

Gaston et al. (2006) propose to use test purposes on the symbolic execution (King, 1975) of the specification rather than on the specification directly. The symbolic execution of the specification leads to a symbolic unfolding of a specification's behavior. A test purpose is then a collection of some finite paths of the symbolic execution.

Test Purposes for real-time systems have been considered by David et al. (2008b). They propose to used Timed CTL formulas as test purposes. Timed CTL (Alur et al., 1990) is an extension of CTL (Computational Tree Logic (Clarke et al., 1986)) which adds timing constraints to the traditional operators of CTL. For example, within Timed CTL one can specify that along some computation path a proposition p becomes true within 5 time units. Test case generation is conducted by synthesizing a winning strategy, i.e. a function that can state whether to send a particular stimuli or to wait for an response from the system under test. Test execution is done incrementally by constantly consulting the winning strategy and the specification in order to select the proper action. If there is a violation of the specification with respect to **tioco** then the test execution fails, otherwise the implementation passes the test run.

However, this approach is limited to cases where a winning strategy exists. Such a strategy does not exist if a system may leave the test purpose, i.e. the system may force the tester to a path where the test purpose will never be satisfied. This limitation has been relaxed by David et al. (2008a). This work shows how a winning strategy may be calculated if the system is willing to cooperate to some extent. Test execution may then also end in an inconclusive verdict, which indicates that the system has not cooperated in order to satisfy the test purpose.

Coverage based

Coverage based testing has a long tradition in software engineering; classical books on software testing, e.g., Myers (1979), describe different well-known coverage criteria. Coverage criteria for logical expressions (Ammann et al., 2003) lend themselves not only to test suite analysis, but also to test case generation. For example, model checkers have been used to automatically generate test cases that satisfy coverage criteria (Gargantini and Riccobene, 2001; Hong et al., 2002; Rayadurgam and Heimdahl, 2001).

Coverage based testing on the level of labeled transition systems has been explored by Huo and Petrenko (2009). Often coverage criteria are not applied to the labeled transition system directly, but to the high-level specification. One specification that is commonly used for input-output conformance testing, and which has also been chosen for this thesis, is LOTOS (ISO, 1989).

Coverage based testing of LOTOS specifications has been considered by van der Schoot and Ural (1995). They presented a technique for test case generation with respect to the definition and the use of variables. They use data flow graphs to identify the relevant traces. The feasibility of these traces is then verified by using guided interference rules on the LOTOS specification. However, they derive linear test cases and only address test sequence selection. The test sequences comprise symbolic values which have to be instantiated during test case execution.

Cheung and Ren (1993) define operational coverage for LOTOS specifications, i.e., coverage criteria that aim to reflect the characteristics of LOTOS operations. Furthermore, they propose an algorithm that derives an executable test sequence from a given specification with respect to their coverage criteria. Their algorithm is based on a Petri-net representation of the LOTOS specification.

Amyot and Logrippo (2000) use a probe insertion technique to measure structural coverage of LOTOS specifications. Their probe insertion strategy allows one to reason about the coverage of a test suite for a LOTOS specification. Furthermore, they present an optimization that reduces the number of needed probes. However, they only considered action coverage. Furthermore, their approach deals with measuring coverage only. They do not consider generating test cases with respect to their inserted probes.

Fault based

Fault-based testing, also known as mutation testing, is a popular technique when using model-checkers for test case generation and has been proposed by Ammann et al. (1998). While model-checker based mutation testing was limited to deterministic models recent research (Boroday et al., 2007; Okun et al., 2002; Fraser and Wotawa, 2007) allows the application of model-checkers to non-deterministic models.

Petrenko and Yevtushenko (2005) showed how to use partial, non-deterministic finite state machines (FSM) for mutation based test case generation. This work makes FSM based testing more amenable in industrial applications where specifications are rarely deterministic and complete.

A work that considers the combination of fault-based testing and test purposes has been done by Petrenko et al. (2004). The used models are extended finite state machines (EFSM). The authors consider one single fault-type where an implementation is in a wrong post-state after applying a particular test sequence. The test purpose is given in terms of configurations of the EFSM denoting suspicious implementation states. That is, the test purpose describes outputs that should be avoided. The authors also consider limiting the length of the test sequence.

Fault-based conformance testing based on LOTOS specifications has been investigated by Aichernig and Corrales Delgado (2006). They propose to use a discriminating sequence extracted from a mutant and the original specification as a test purpose to the TGV tool. This test purpose leads to a test case that fails if the mutant has been implemented.

Briones et al. (2006) proposed an approach for combining coverage and fault-based testing of labeled transition systems. They propose to assign weights according to the severity of possible faults to the outputs of a labeled transition system. In other words, they assign every state of a model the weights for different (faulty) outputs. These weighted fault models are represented in terms of fault automata. Then they use different algorithms to generate test cases covering the fault automata.

Random and Heuristics

The very first test case selection strategy for **ioco** testing was random testing (Tretmans, 1996) which has been implemented by Tretmans and Brinksma (2003) in the TORX tool. Basically, the implemented algorithm takes a specification and makes in any step one of the following choices: (1) terminate the test case with a pass verdict; (2) send a stimuli to the implementation (if allowed by the specification); (3) wait for a response from the implementation (if allowed by the specification).

While the probabilities for choosing between (1), (2) and (3) have been set to one third each, Feijs et al. (2000) propose to use different probabilities for each of the three possible choices. Their idea is based on the observation that if there are *n* inputs and *m* outputs the likelihood to generate a particular trace can be increased by adjusting the choice probabilities. Goga (2003) evaluated this approach on the conference protocol with the result that probabilistic TORX outperformed TORX in terms of the length of failure revealing traces.

While the approach of Feijs et al. (2000) only changes the probabilities of the three different options for continuing a test run, Burguillo-Rial et al. (2002) propose to use a risk-based approach to select specific transitions of the specification. Basically, Burguillo-Rial et al. (2002) rely on a risk measurement function provided by the test engineer and on the execution counts of transitions to steer the test case generation process towards the more risky branches of the specification.

Pyhälä and Heljanko (2003) propose to guide the random based test case selection process of TORX by means of specification coverage. During random testing Pyhälä and Heljanko keep track of already covered parts of the specification. When selecting new transitions they use actions that have not been covered yet. If there is no such action at the current state, they recommend using bounded model-checking (Biere et al., 2003) to find an uncovered action within a certain bound. Then the first action of the trace leading to the uncovered action, i.e. the first action of the model-checker's counter-example, is used for continuing testing. If the model-checker does not find a counter-example, Pyhälä and Heljanko (2003) fall back to random selection.

Lestiennes and Gaudel (2002) use testing hypotheses (Gaudel, 1995) to restrict the possibly infinite set of test cases to a finite number of test cases. First they use the assumptions and properties of the **ioco** conformance relation to remove meaningless test cases from the set of possible test cases. For example, due to input-enabledness of implementations test cases that terminated with a pass verdict after providing an input are not relevant. Second, they use regularity hypotheses to bound the length of a test case and uniformity hypotheses to define equivalence classes on the values of actions.

A similar approach has been presented by Feijs et al. (2002) who also consider the restriction of the branching degree of nodes (uniformity hypotheses). Furthermore, Feijs et al. claim that only a low number of iterations through a loop of a test case reveal different behavior of implementations (regularity hypotheses).

Timed testing by the use of TORX has been considered by Bohnenkamp and Belinfante (2005), while a symbolic version of TORX's testing algorithm has been proposed by Frantzen et al. (2004).

3.4 LOTOS- A modelling language

The input-output conformance relation is based on labeled transition systems. While LTSs are suitable models for representing test cases, specifications, and implementations, one does not want to write specifications in terms of labeled transition systems. Therefore one needs a specification language having the semantics of labeled transition systems but providing a simple and powerful syntax for writing large specifications. One such specification language is the language of temporal ordering specification (LOTOS).

LOTOS is an ISO standard (ISO, 1989) and comprises two components: The first is based on the Calculus of Communication Systems (Milner, 1980) (CCS) and deals with the behavioral description of a system, which is given in terms of processes, their behavior, and their interactions. The second component of LOTOS specifications is used to describe data structures and value expressions, and is based on the abstract data type language ACT ONE (Ehrig et al., 1983).

Because LOTOS is a powerful process algebra and there is a mature tool box (Garavel et al., 2002), i.e. the CADP toolbox, that provides a model-checker (evaluator (Mateescu and Sighireanu, 2000)), a handy programming interface for accessing a specification's underlying LTS (Garavel, 1998), an equivalence checker (bisimulator (Mateescu and Oudot, 2008)), and a test case generator (TGV (Jard and Jéron, 2005)), we rely on LOTOS as modelling language.

This section gives a brief introduction into LOTOS. For additional tutorials we refer to Bolognesi and Brinksma (1987) and to Turner (1989, 1993).

The basic elements of a LOTOS specification are processes with a certain behavior expressed in terms of actions. An action is an expression over a process's gates, possibly equipped with values. Table 3.1 lists some of the main elements used to compose the behavior of a process. As listed in this table, an action is basically one of four expressions. There is the internal action *i*, which denotes an unobservable event. *i* corresponds to a τ transition within an LTS. There is the action without any parameters, i.e. only the name of a process's gate. Such an action expresses that the process offers this action for communication. Action statements may offer values, i.e., `g !value` or actions may read values, i.e., `g ?value:Type`.

Sequential composition is denoted by `action; behavior`, i.e., first the action is offered for communication and then the statements of the following (`behavior`) block are executed.

The actions within a guarded behavior are only enabled if the guard evaluates to true.

A choice between two behaviors (`behavior1 [] behavior2`) expresses that the very first action of `behavior1` and of `behavior2` are offered for communication. Once one of the offered actions has been chosen, the composed process behaves like `behavior1` or `behavior2`. Basically, `[]` expresses an external choice, i.e., the environment can chose between the offered actions. Internal choices, i.e., choices where a process itself decides on the offered actions, can be implemented using the special action *i*. *i* represents an internal transition, i.e., *i* results in a τ-labeled transition within the underlying LTS.

Example 3.16. Figure 3.9 illustrates the difference between internal and external choices in LOTOS and in the underlying labeled transition systems. The specification and the LTS shown on the left depict an

Table 3.1: Excerpt of LOTOS behavior elements.

Syntax	Meaning
`i`	internal action
`g`	action, i.e., a process gate
`g !value`	action offering a value
`g ?value:Type`	action reading a value
`action; behavior`	action followed by a behavior
`[guard] -> behavior`	guarded behavior
`behavior1 [] behavior2`	choice
`behavior1 || behavior2`	synchronization
`behavior1 ||| behavior2`	interleaving
`behavior1 |[gate1,..]| behavior2`	partial synchronization
`behavior1 [> behavior2`	disabled by second behavior
`behavior1 >> behavior2`	first enables second
`exit`	exit
`stop`	stop, inaction
`proc[gate,..](val,..)`	process instantiation
`( behavior )`	grouping

```
process P1[a, b] : exit :=
  a; exit [] b; exit
endproc
```

```
process P2[a, b] : exit :=
  i; a; exit [] i; b; exit
endproc
```

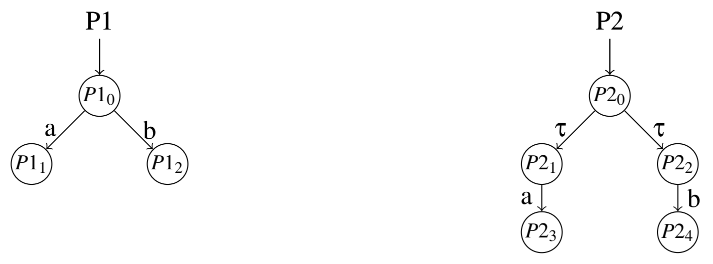


Figure 3.9: External and internal choices.

external choice between the actions a and b. That is, in the state $P1_0$ the environment can chose whether to synchronize on a or on b. In contrast to that, the process P2 and its LTS depict an internal choice between a and b. That is, the system may internally decide to move to state $P2_1$ or to state $P2_2$. In any case only one action is offered for communication. □

LOTOS supports three different operators to express parallel composition of processes, i.e., `||`, `|||`, and `|[...]|`. The interleaving expression $B_1|||B_2$ states that the two behaviors B_1 and B_2 are executed without any synchronization, i.e. the actions may be executed in an arbitrary order. For example, let a_1, a_2 and b_1 be actions, then the specification $(a_1;a_2)|||(b_1)$ has the following observable behaviors: $\langle a_1, a_2, b_1\rangle$, $\langle a_1, b_1, a_2\rangle$, $\langle b_1, a_1, a_2\rangle$. On the contrary, partial synchronization $(B_1|[g_1, \ldots, g_n]|B_2)$ expresses synchronization only on the specified gates $g_1, \ldots, g_n$. A synchronized action can only be executed if both synchronized behavioral blocks offer the action,e.g. let a_1, a_2 and b_1 be actions, then the specification $(a_1;a_2)|[a_2]|(a_2;b_1)$ has only the following action sequence: $\langle a_1, a_2, b_1\rangle$. Finally, full synchronization, i.e. $B_1||B_2$, requires that all actions common to both, B_1 and B_2, are synchronized.

The enabling operator (») within a LOTOS specification states that a successful execution of the first process enables the following behavior block. In contrast to that, the disabling operator (`[>`) states that any action from the second behavior disables the execution of the first behavior. Finally, the behavioral part of

```
1  type Stack is NaturalNumber
2    sorts
3      Stack
4    opns
5      nil  : -> Stack
6      push : Nat, Stack -> Stack
7      pop  : Stack -> Stack
8      top  : Stack -> Nat
9      size : Stack -> Nat
10   eqns
11     forall st: Stack, n: Nat
12       ofsort Stack
13         pop(push(n,st)) = st;
14       ofsort Nat
15         size(nil) = 0;
16         size(push(n,st)) = Succ(size(st));
17         top(push(n,st)) = n;
18 endtype
```

Figure 3.10: A simple stack data type providing typical stack operations.

LOTOS supports the definition of processes (for an example see Figure 3.9), instantiation of processes and grouping of behavior.

An abstract data type is given in terms of sorts, operations and equations.

Example 3.17. Figure 3.10 illustrates the basic elements of data type definitions. The shown data type comprises the base element nil (Line 5), which denotes an empty stack, and the four prefix operators `push`, `pop`, `top`, and `size`. For example, the signature of the `push` operator on Line 6 shows that this operator takes a natural number (Nat) and a stack and returns a new stack. The equation part of this data type definition states that for any stack *st* and for any natural number *n*, popping a previously pushed element of a stack *st* results in *st* (Line 13). □

3.4.1 Example LOTOS Specification of a Stack Calculator

Figure 3.11 shows a LOTOS specification of a simple stack calculator, which will be used to illustrate the concepts and approaches of this thesis.

The specification uses both behavioral description elements and data type elements. The stack data type is specified in Figure 3.10. For the sake of brevity we do not show the data type definitions for Boolean elements and for natural numbers here; our specification simply relies on the types "Boolean" and "NaturalNumber" stated in the appendix of the LOTOS ISO (1989) standard. In addition, we omit the definition of our "Operator" type, which only provides the two base elements `add` and `display` and comparison operators for those two elements.

The specification of the stack calculator uses the two gates `ui` for user input and `out` for system output and consists of two processes. The main process (Lines 5-25) takes the current stack *s* as argument and reads either an operator (Line 7) or an operand (Line 24), i.e., a natural number, from the user. If the user enters a natural number, the number is pushed onto the stack and the execution of the main process is continued recursively (Line 24).

If the user enters an operator, the specification checks which operator has been entered. If the `display` operator has been selected and the stack is not empty (Line 9) then the specification calls the `DisplayStack` process and the `Main` process subsequently without changing the stack (Line 10).

If the entered operator equals `add` and the size of the stack is greater than or equal to two (Line 13) then we recursively continue with a stack where the two top elements are replaced by their sum (Line 14).

If the stack size does not meet the requirements of the entered operator (Lines 17 and 18) then the calculator issues an error and continues without changing the stack (Line 19).

The `DisplayStack` process (Lines 27-33) takes a stack *s* and displays the content of the stack using the `out` gate.

The overall behavior of the calculator's specification (Line 3) is composed of instantiating the main process with an empty stack (i.e., nil) with the possibility to disable operation of the main process at any time using a `quit` command.

```
1  specification StackCalc [ui,out]:exit
2  behavior
3    Main[ui,out](nil) [> ui !quit; exit
4  where
5    process Main[ui, out](s:Stack):noexit:=
6     (
7      ui ? op : Operator;
8       (
9        [(op eq display) and (size(s) gt 0)]→(
10         DisplayStack[out](s) >> Main[ui,out](s)
11       )
12       []
13       [(op eq add) and (size(s) ge Succ(Succ(0)))]→(
14         Main[ui,out](push(top(s)+top(pop(s)), pop(pop(s))))
15       )
16       []
17       [((op eq add) and (size(s) lt Succ(Succ(0))))
18        or ((op eq display) and (size(s) eq 0))]→ (
19         out !error; Main[ui,out](s)
20       )
21      )
22     )
23     []
24     ( ui ? num: Nat; Main[ui,out](push(num,s)) )
25    endproc
26
27    process DisplayStack[out](s:Stack):exit:=
28     [size(s) eq Succ(0)] →
29       out !top(s); exit
30     []
31     [size(s) gt Succ(0)] →
32       out !top(s); DisplayStack[out](pop(s))
33    endproc
34 endspec
```

Figure 3.11: LOTOS specification of a simple stack calculator.

Chapter 4

Unifying Input-Output Conformance

Parts of the contents of this chapter have been published in (Weiglhofer and Aichernig, 2008b) and (Weiglhofer and Aichernig, 2008a).

Computer science is a widely studied field of research with many different branches that are worth being investigated. In the last few decades of research various findings have led to different concepts and paradigms. For example, there is a considerable number of programming paradigms (e.g. functional, object-oriented, logical) that have arisen in the past years. Each research area establishes its own formalism for reasoning over the various concepts.

Hoare and He (1998) aim to unify the different concepts and paradigms by providing a global framework, i.e. the Unifying Theories of Programming (UTP). This framework allows uniform reasoning for all embedded concepts. While the original UTP theory deals with an algebra of programs, concurrency, communication and high order programming, UTP has been extended during the past years. For example, Cavalcanti et al. (2006) added a theory for pointers while He and Sanders (2006) added a theory for probability.

Correctness in UTP is handled by the use of logical implication: Given a specification S and a program P, where S and P have the same alphabet of variables, then P is correct with respect to S iff

$$\forall v, w, \ldots \bullet P \Rightarrow S$$

$v, w, \ldots$ denote all the variables of the alphabet of P and S. As usual (e.g. see (Dijkstra and Scholten, 1990)) we will use $[P \Rightarrow S]$ as a shorthand for $\forall v, w, \ldots \bullet P \Rightarrow S$.

This notion of correctness can for example be used for step-wise refinement. Starting from an abstract specification one refines the specification until a concrete executable program is derived. However, in the context of black-box testing refinement is too strong. For black-box testing one has to distinguish between inputs and outputs and one has to deal with non-determinism and incomplete specifications.

In the following we recast the theory of input-output conformance testing in the UTP framework. The benefits of this reformulation can be summarized as follows:

- Instead of describing the assumptions of **ioco** informally, the UTP formalization presents the underlying assumptions as unambiguous healthiness conditions and by adopted choice operators over reactive processes;
- A UTP formalization naturally relates **ioco** and refinement in one theory;
- The denotational version of **ioco** enables formal, machine checkable, proofs.

- Due to the predicative semantics of UTP, test case generation based on the presented theory can be seen as a satisfiability problem. This facilitates the use of modern sat modulo theory techniques (e.g. (Dutertre and de Moura, 2008)) for test case generation.
- Finally, the UTP version of **ioco** broadens the scope of **ioco** to specification languages with UTP semantics, e.g. to generate test cases from Circus (Oliveira et al., 2007) specifications. Hence our work enriches UTP's reactive processes with a practical testing theory.

4.1 Unifying Theories of Programming

In this section we briefly summarize the UTP concepts needed in this thesis. For tutorials and additional information we refer to Woodcock and Cavalcanti (2004) and to Cavalcanti and Woodcock (2004).

The UTP framework uses an alphabetised version of Tarski's relational calculus (Tarski, 1941) to reason over the various concepts. UTP uses alphabetized predicates, i.e. alphabet-predicate pairs. All free variables of the predicate are members of the alphabet. The relations are predicates over initial and intermediate or final observations. UTP uses undecorated variables (e.g. x, y, z) to denote initial observations and dashed variables (e.g. x', y', z') to represent intermediate or final observations.

The alphabet of a predicate P is denoted by αP. This alphabet is divided into before-variables (undecorated), i.e. $in\alpha P$, and into after-variables (dashed), i.e. $out\alpha P$.

Definition 4.1 (Relation) *A relation is a pair* $(\alpha P, P)$*, where P is a predicate containing no free variables other than those in* αP*, and* $\alpha P = in\alpha P \cup out\alpha P$ *where* $in\alpha P$ *is a set of undashed variables standing for initial values, and* $out\alpha P$ *is a set of dashed variables standing for final values.*

The alphabetized predicates are combined by the use of standard predicate calculus operators (e.g. $\wedge$, $\vee$). However, the definitions of these operators have to be extended by an alphabet specification. For instance, in the case of a conjunction the alphabet is the union of the conjoined predicates: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$.

While UTP formalizes many programming concepts we briefly summarize the ones needed in this thesis. One of the most important concepts is the conditional. While the conditional is usually written as **if** b **then** P **else** Q, Hoare and He choose an infix notation.

Definition 4.2 (Conditional)

$$\begin{aligned} P \lhd b \rhd Q &=_{df} (b \wedge P) \vee (\neg b \wedge Q), \textit{ if } \alpha b \subseteq \alpha P = \alpha Q \\ \alpha(P \lhd b \rhd Q) &=_{df} \alpha P \end{aligned}$$

The relevant laws for the conditional statement are the following (Hoare and He, 1998):

L1 $P \lhd b \rhd P = P$ (conditional idempotent)

L2 $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$ (conditional symmetry)

L3 $(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$ (conditional associativity)

L4 $P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$ (conditional distributivity)

L5 $P \lhd true \rhd Q = P = Q \lhd false \rhd P$ (conditional unit)

L6 $(P \wedge Q) \lhd b \rhd (R \wedge S) = (P \lhd b \rhd Q) \wedge (R \lhd b \rhd S)$ (mutual distribution)

Another commonly used concept is sequential composition, often denoted by a semicolon. The sequential composition of two program statements expresses that the program starts with the first statement and after successful termination the second statement is executed. The final state of the first statement serves as initial state of the second statement, however, this intermediate state cannot be observed. All this is formalized in UTP using existential quantification.

Definition 4.3 (Sequential Composition)

$$\begin{aligned} P(v,v');Q(v,v') &=_{df} \exists v_0 \bullet P(v,v_0) \wedge Q(v_0,v') \\ in\alpha(P(v,v');Q(v,v')) &=_{df} in\alpha P \\ out\alpha(P(v,v');Q(v,v')) &=_{df} out\alpha Q \end{aligned}$$

Note that the above definition uses one of the two substitution syntaxes. That is, some or all of the free variables of a predicate are listed in brackets, e.g. $P(x)$. Let f be a list of expressions, then $P(f)$ denotes the result obtained by replacing every variable in x with the expression at the corresponding position in f.

The second, more explicit, notation used within UTP is $P[e/x]$ which stands for the result obtained by substituting every variable x in P with the expression e.

Sequential composition obeys the following two important laws (Hoare and He, 1998):

L1 $P;(Q;R) = (P;Q);R$ (; associativity)

L2 $(P \lhd b \rhd Q);R = (P;R) \lhd b \rhd (Q;R)$ (;-conditional left distributivity)

The third UTP operator that we need throughout this chapter is the operator expressing a non-deterministic choice. If P and Q are both predicates describing a particular program behavior, then $P \sqcap Q$ expresses that either P or Q is executed. Note that this operator does not give a clue on which of the two behavior is chosen.

Definition 4.4 (Non-deterministic choice)

$$\begin{aligned} P \sqcap Q &=_{df} P \vee Q, \textit{ provided that } \alpha P = \alpha Q \\ \alpha(P \sqcap Q) &=_{df} \alpha P \end{aligned}$$

The relevant laws for the non-deterministic choice operator are the following (Hoare and He, 1998):

L1 $P \sqcap Q = Q \sqcap P$ ($\sqcap$ symmetry)

L2 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ ($\sqcap$ associativity)

L3 $P \sqcap P = P$ ($\sqcap$ idempotent)

L4 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$ ($\sqcap$ distributivity)

L5 $P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$ (conditional-$\sqcap$ distributivity)

L6 $(P \sqcap Q);R = (P;R) \sqcap (Q;R)$ (;-$\sqcap$ left distributivity)

L7 $P;(Q \sqcap R) = (P;Q) \sqcap (P;R)$ (;-$\sqcap$ right distributivity)

L8 $P \sqcap (Q \lhd b \rhd R) = (P \sqcap Q) \lhd b \rhd (P \sqcap R)$ ($\sqcap$-conditional distributivity)

One concept formalized within the Unifying Theories of Programming is the theory of communication in terms of reactive processes. Reactive processes with different types of synchronization between them is a widely studied field of research. Different notations like CSP (Hoare, 1978), CCS (Milner, 1989), ACP (Bergstra and Klop, 1985), LOTOS (see Section 3.4), and others have been proposed to formally reason about communication between processes.

As testing can be seen as an interaction between a tester and an implementation under test (IUT), i.e. as a communication between two processes, our denotational version of the input-output conformance relation is based on reactive processes.

4.1.1 Reactive Processes

A reactive process with respect to the unified theories of programming is defined as follows:

Definition 4.5 (Reactive process) *A reactive process P is one which satisfies the healthiness conditions **R1** (a reactive process does not undo past events), **R2** (changing the preceding trace does not change the behavior of the process), **R3** (if a process is asked to start in a waiting state of another process, it does not do anything) where*

$$\begin{aligned}
\textbf{\textit{R1}}(X) &=_{df} X \wedge (tr \leq tr') \\
\textbf{\textit{R2}}(X(tr,tr')) &=_{df} \bigsqcap_{s} X(s, s^\frown(tr' - tr)) \\
\textbf{\textit{R3}}(X) &=_{df} \mathbb{I} \lhd wait \rhd X
\end{aligned}$$

The alphabet of P consists of the following:

- *$\mathcal{A}$, the set of events in which it can potentially engage.*
- *$tr : \mathcal{A}^*$, the sequence of events which have happened up to the time of observation.*
- *$ref : \mathcal{P}\mathcal{A}$, the set of events refused by the process during its wait, where $\mathcal{P}X$ is the powerset of X.*
- *$wait$: Bool, which distinguishes its waiting states from its terminated states.*
- *ok, ok' : Bool, indicating start and termination of a process*

The skip predicate ($\mathbb{I}$) is defined as by Hoare and He (1998):

$$\mathbb{I} =_{df} \neg ok \wedge (tr \leq tr') \vee ok' \wedge (tr' = tr) \wedge \cdots \wedge (wait' = wait)$$

A process offering a single event $a \in \mathcal{A}$ for communication is expressed in terms of $do_{\mathcal{A}}(a)$, where

$$\begin{aligned}
do_{\mathcal{A}}(a) &=_{df} \Phi(a \notin ref' \lhd wait' \rhd tr' = tr^\frown \langle a \rangle) \\
\Phi &=_{df} \mathbf{R} \circ and_B = and_B \circ \mathbf{R}, \qquad B =_{df} ((tr' = tr) \wedge wait' \vee (tr < tr')) \\
\mathbf{R} &=_{df} \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}
\end{aligned}$$

Using the notion from above we can now express the denotational semantics of various statements.

Example 4.1. Suppose we have a process that is composed of the sequential composition of the two actions a and b, then the labeled transition system representing this process looks as follows:

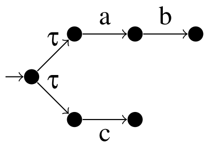
$$\begin{tikzcd}
{} \arrow[r] & \bullet \arrow[r, "a"] & \bullet \arrow[r, "b"] & \bullet
\end{tikzcd}$$

In terms of UTP the behavior of this process is given by the following derivation:

$$\begin{aligned}
& a;b = && \text{(R3)} \\
& = \mathbb{I} \lhd wait \rhd (a;b) && \text{(def. of } d_{\mathcal{A}}(a)\text{)} \\
& = \mathbb{I} \lhd wait \rhd (\Phi(a \notin ref' \lhd wait' \rhd tr' = tr^\frown \langle a \rangle);b) && \text{(R3(b) and ; def)} \\
& = \mathbb{I} \lhd wait \rhd \exists w_0 \bullet (a \notin ref' \wedge tr' = tr) \lhd w_0 \rhd (tr' = tr^\frown \langle a \rangle \wedge do_{\mathcal{A}}(b)) && \text{(def. of } do_{\mathcal{A}}(b)\text{)} \\
& = \mathbb{I} \lhd wait \rhd \exists w_0 \bullet (a \notin ref' \wedge tr' = tr) \lhd w_0 \rhd && \text{(cond assoc.)} \\
& \qquad ((tr' = tr^\frown \langle a \rangle \wedge b \notin ref') \lhd wait' \rhd (tr' = tr^\frown \langle a,b \rangle)) && \\
& = \mathbb{I} \lhd wait \rhd \exists w_0 \bullet ((a \notin ref' \wedge tr' = tr) \lhd w_0 \rhd && \text{(pred. calc. and R3)} \\
& \qquad (tr' = tr^\frown \langle a \rangle \wedge b \notin ref')) \lhd w_0 \wedge wait' \rhd && \\
& \qquad (tr' = tr^\frown \langle a,b \rangle) && \\
& = (a \notin ref' \wedge tr' = tr) \vee (tr' = tr^\frown \langle a \rangle \wedge b \notin ref') \lhd wait' \rhd (tr' = tr^\frown \langle a,b \rangle) &&
\end{aligned}$$

As the final line of this derivation shows, the process's behavior is divided into the waiting ($wait'$) and the non-waiting ($\neg wait'$) behavior. If the process is waiting for communication, i.e. $wait' = true$, then either no event has happened ($tr' = tr$) and the process does not refuse a, or a has happened ($tr' = tr^\frown\langle a\rangle$) and the process offers b for communication ($b \notin ref'$). If the process is not waiting anymore the actions a and b have been done ($tr' = tr^\frown\langle a,b\rangle$). □

Example 4.2. The following LTS shows a process expressing a non-deterministic choice between the actions a and c where after the action a b is executed.



In terms of UTP's reactive processes this process is expressed as follows.

$$
\begin{aligned}
&(a;b) \sqcap c = && \text{(def. of } \sqcap\text{)}\\
&= \mathbf{R3}((a;b) \vee c) && \text{(Example 1)}\\
&= \mathbf{R3}(((a \notin ref' \wedge tr' = tr) \vee (tr' = tr^\frown\langle a\rangle \wedge b \notin ref') \lhd wait' \rhd && \text{(def of } do_{\mathcal{A}}(c))\\
&\qquad (tr' = tr^\frown\langle a,b\rangle)) \vee c)\\
&= \mathbf{R3}(((a \notin ref' \wedge tr' = tr) \vee (tr' = tr^\frown\langle a\rangle \wedge b \notin ref') \lhd wait' \rhd && \text{(simplification)}\\
&\qquad (tr' = tr^\frown\langle a,b\rangle)) \vee\\
&\qquad \Phi(c \notin ref' \lhd wait' \rhd tr' = tr^\frown\langle c\rangle))\\
&= ((a \notin ref' \vee c \notin ref') \wedge tr' = tr) \vee (tr' = tr^\frown\langle a\rangle \wedge b \notin ref') \lhd wait' \rhd\\
&\qquad (tr' = tr^\frown\langle c\rangle \vee tr' = tr^\frown\langle a,b\rangle)
\end{aligned}
$$

□

Note that the process follows an internal choice semantics, i.e. offering communication of *a or b* at the beginning to the environment.

A special process is the deadlock process which denotes inaction:

$$\mathfrak{d} =_{df} \mathbf{R3}(tr' = tr \wedge wait')$$

Note that there is a name clash between the symbol used for representing quiescence and the deadlock symbol. Thus, in this thesis we use $\mathfrak{d}$ as UTP's deadlock symbol and δ for denoting quiescence.

The input-output conformance relation distinguishes between inputs and outputs. Hence, the alphabet $\mathcal{A}$ of a process consists of two disjoint sets $\mathcal{A} = \mathcal{A}_{in} \cup \mathcal{A}_{out}$. In addition, we will also differentiate between refused inputs $ref_{in} =_{df} ref \cap \mathcal{A}_{in}$ and refused outputs $ref_{out} =_{df} ref \cap \mathcal{A}_{out}$. Thus, also refusals form a partition: $ref_{in} \cap ref_{out} = \emptyset$ and $ref = ref_{in} \cup ref_{out}$. Note that we use ? and ! to indicate inputs and outputs for processes. For example, a process having as input alphabet $\mathcal{A}_{in} = \{1\}$ and as output alphabet $\mathcal{A}_{out} = \{c\}$ is written as $do_{\mathcal{A}}(?1); do_{\mathcal{A}}(!c)$.

4.2 IOCO Specifications

For technical reasons, that is the computability of particular sets during the test case generation, the reactive processes used in the **ioco** framework need to satisfy an additional healthiness condition. The processes need to be strongly responsive, i.e. processes do not comprise livelocks. If there is a livelock a process may execute while it never offers communication. Hence, the healthiness condition for specifications excludes livelocks:

IOCO1 $\quad P = P \wedge (ok \Rightarrow (wait' \vee ok'))$

As a function, **IOCO1** is defined as $\mathbf{IOCO1}(P) = P \wedge (ok \Rightarrow (wait' \vee ok'))$. It is an idempotent.

Lemma 4.1 (IOCO1-idempotent)

$$\boldsymbol{IOCO1} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1}$$

$\mathbb{I}$ is **IOCO1** healthy.

Lemma 4.2 ($\mathbb{I}$-IOCO1-healthy)

$$\boldsymbol{IOCO1}(\mathbb{I}) = \mathbb{I}$$

The healthiness condition **IOCO1** is independent from **R1**, **R2**, and **R3**, i.e. they commute.

Lemma 4.3 (commutativity-IOCO1-R1)

$$\boldsymbol{IOCO1} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{IOCO1}$$

Lemma 4.4 (commutativity-IOCO1-R2)

$$\boldsymbol{IOCO1} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{IOCO1}$$

Lemma 4.5 (commutativity-IOCO1-R3)

$$\boldsymbol{IOCO1} \circ \boldsymbol{R3} = \boldsymbol{R3} \circ \boldsymbol{IOCO1}$$

Furthermore, composing two **IOCO1** healthy processes using sequential composition, gives us another **IOCO1** healthy process.

Lemma 4.6 (closure-;-IOCO1)

$$\boldsymbol{IOCO1}(P;Q) = P;Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO1} \textit{ and } \boldsymbol{R3} \textit{ healthy}$$

Within the theory of Tretmans (1996) quiescence denotes the absence of outputs and the absence of internal actions. Quiescence is encoded by the presence of a particular action δ. Although, quiescence can be classified by $wait'$ and ref' it is necessary to include δ into the traces of processes (see Example 4 $\neg l$ **ioco** k).

Since **ioco** uses traces containing quiescence we need to include δ in the traces of our processes. Thus, we extend set of events for reactive processes $\mathcal{A}$ by δ. In the sequel we use the following abbreviation $\mathcal{A}_\delta =_{df} \mathcal{A} \cup \{\delta\}$. A UTP process is quiescent after a particular trace iff either it has finished its execution or it refuses to do any output action

$$quiesence =_{df} \neg wait' \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref'$$

Quiescent communication is expressed in terms of $do_{\mathcal{A}}^{\delta}$, which adds quiescence (δ) to the traces and to the refusal set of a process.

Definition 4.6 (Quiescent communication) *Let $a \in \mathcal{A}$ be an action of a process's alphabet, then*

$$do^{\delta}_{\mathcal{A}}(a) =_{df} \begin{cases} \Phi^i(do_{\mathcal{A}}(a)) & \text{if } a \in A_{out} \\ \Phi^i(\{\delta, a\} \not\subseteq ref' \wedge tr' - tr \in \delta^* \lhd wait' \rhd tr' - tr \in \delta^{*\frown}\langle a \rangle) & \text{if } a \in A_{in} \end{cases}$$

where $\Phi^i =_{df} \boldsymbol{IOCO1} \circ \boldsymbol{R} \circ and_B$

Consider the case where a is an input action, i.e., $a \in A_{in}$: In the case of $wait'$ the process $do^{\delta}_{\mathcal{A}}(a)$ allows an arbitrary number of δ events (see the δ-loops in Fig. 3.2). After termination the event a has happened preceded by an arbitrary - possible empty sequence - of quiescence events, i.e. $tr' - tr \in \delta^{*\frown}\langle a \rangle$. Note that $\delta^{*\frown}\langle a \rangle$ denotes the set of events where every element of δ^* is concatenated with $\langle a \rangle$.

The possible occurrence of δ events is formalized as follows:

IOCO2	$P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$
IOCO3	$P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s^\frown\delta^*)))$

The antecedence $\neg wait$ is necessary due to the same reasons as in **R3**, i.e. if a process is asked to start in a waiting state of its predecessor the process should not change anything.

As functions, **IOCO2** and **IOCO3** can be defined as **IOCO2**$(P) = P \wedge (wait' \Rightarrow (\delta \notin ref' \iff quiescence))$ and **IOCO3**$(P) = P \wedge (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr \in s^\frown\delta^*))$, respectively. Both functions, **IOCO2** and **IOCO3** are an idempotent.

Lemma 4.7 (IOCO2-idempotent)

$$\boldsymbol{IOCO2} \circ \boldsymbol{IOCO2} = \boldsymbol{IOCO2}$$

Lemma 4.8 (IOCO3-idempotent)

$$\boldsymbol{IOCO3} \circ \boldsymbol{IOCO3} = \boldsymbol{IOCO3}$$

IOCO2 and **IOCO3** are independent, i.e. they commute.

Lemma 4.9 (commutativity-IOCO2-IOCO3)

$$\boldsymbol{IOCO2} \circ \boldsymbol{IOCO3} = \boldsymbol{IOCO3} \circ \boldsymbol{IOCO2}$$

In addition, both **IOCO2** and **IOCO3** are independent from **R1**, **R2** and **IOCO1**.

Lemma 4.10 (commutativity-IOCO2-R1)

$$\boldsymbol{IOCO2} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{IOCO2}$$

Lemma 4.11 (commutativity-IOCO2-R2)

$$\boldsymbol{IOCO2} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{IOCO2}$$

Lemma 4.12 (commutativity-IOCO2-IOCO1)

$$\boldsymbol{IOCO2} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{IOCO2}$$

Lemma 4.13 (commutativity-IOCO3-R1)

$$\boldsymbol{IOCO3} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{IOCO3}$$

Lemma 4.14 (commutativity-IOCO3-R2)

$$\boldsymbol{IOCO3} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{IOCO3}$$

Lemma 4.15 (commutativity-IOCO3-IOCO1)

$$\boldsymbol{IOCO3} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{IOCO3}$$

Lemma 4.16 (commutativity-IOCO3-IOCO2)

$$\boldsymbol{IOCO3} \circ \boldsymbol{IOCO2} = \boldsymbol{IOCO2} \circ \boldsymbol{IOCO3}$$

By introducing the observability of quiescence we need to change the definition of the skip ($\mathbb{I}$) element: Processes always need to respect the properties of quiescence. Even in the case of divergence δ can be observed if and only if there is no output. This leads to $\mathbb{I}^\delta$, which is defined as follows.

$$\mathbb{I}^\delta =_{df} \left(\begin{array}{l} \neg ok \wedge (tr \leq tr') \wedge (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)) \wedge \\ (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s ^\frown \delta^*)) \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \\ (v' = v) \end{array} \right)$$

In the above definition the variables v and v' denote the observation vectors, i.e., $v = \{ok, wait, tr, ref\}$ and $v' = \{ok', wait', tr', ref'\}$, respectively.

By introducing a new skip element we also need to change the definition of the healthiness condition **R3**. We will denote this modified healthiness condition as

$$\mathbf{R3}^\delta(P) =_{df} \mathbb{I}^\delta \lhd wait \rhd P$$

Lemma 4.17 (commutativity-IOCO2-R3$^\delta$)

$$\boldsymbol{IOCO2} \circ \boldsymbol{R3}^\delta = \boldsymbol{R3}^\delta \circ \boldsymbol{IOCO2}$$

Lemma 4.18 (commutativity-IOCO3-R3$^\delta$)

$$\boldsymbol{IOCO3} \circ \boldsymbol{R3}^\delta = \boldsymbol{R3}^\delta \circ \boldsymbol{IOCO3}$$

IOCO2 and **IOCO3** form a closure with respect to sequential composition. That is, composing two healthy processes using sequential composition gives again a healthy process.

Lemma 4.19 (closure-;-IOCO2)

$$\boldsymbol{IOCO2}(P;Q) = P;Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO2} \textit{ and } \boldsymbol{R3}^\delta \textit{ healthy}$$

Lemma 4.20 (closure-;-IOCO3)

$$\boldsymbol{IOCO3}(P;Q) = P;Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO3} \textit{ and } \boldsymbol{R3} \textit{ healthy}$$

Since the quiescence event δ encodes the absence of output events it may occur at any time. This is even true for the deadlock process.

Definition 4.7 (Quiescent deadlock)

$$\delta^\delta =_{df} \boldsymbol{R3}^\delta(tr' - tr \in \delta^* \wedge wait')$$

Consequently, the classical deadlock process indicating absolute inactivity does not exist within the **ioco** theory. As the deadlock, also the quiescent deadlock is a left zero for sequential composition:

Lemma 4.21 ($\mathfrak{S}^{\delta}$-left-zero)

$$\mathfrak{S}^{\delta};P = \mathfrak{S}^{\delta}$$

Although quiescence is preserved by sequential composition, we need to redefine internal and external choices in order to preserve the properties of quiescence. Basically, the composition of processes that start with input actions (i.e. the processes are quiescent initially) is quiescent. If one of the two composed processes is not quiescent initially, the composition is not quiescent either.

For our quiescence preserving composition operators ($\sqcap^{\delta}$, $+^{\delta}$) we use an approach similar to parallel by merge (Hoare and He, 1998). The idea of parallel by merge is to run two processes independently and merge their results afterwards. In order to express independent execution we need a relabeling function. Given an output alphabet $\{v'_1, v'_2, \ldots, v'_n\}$, U_l is defined as follows

Definition 4.8 (Relabelling)

$$\begin{aligned}
\alpha U_l(\{v'_1, v'_2, \ldots, v'_n\}) &=_{df} \{v_1, v_2, \ldots, v_n, l.v'_1, l.v'_2, \ldots, l.v'_n\} \\
U_l(\{v'_1, v'_2, \ldots, v'_n\}) &=_{df} (l.v'_1 = v_1) \wedge (l.v'_2 = v_2) \wedge \cdots \wedge (l.v'_n = v_n)
\end{aligned}$$

Independent execution of P and Q is now expressed by relabeling:

Definition 4.9 (Independent execution)

$$P \curlywedge Q =_{df} P;U_0(out\alpha P) \wedge Q;U_1(out\alpha Q)$$

Given the independent execution of two processes and a merge relation we can unfold the definition as follows:

Lemma 4.22 (unfolded-independent-execution)

$$(P \curlywedge Q);M = \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M[v_0, v_1/0.v, 1.v]$$

An internal choice, which takes care of δ within the resulting process, can be defined as follows:

Definition 4.10 (Quiescence preserving internal choice)

$$P \sqcap^{\delta} Q \quad =_{df} \quad (P \curlywedge Q);M_{\sqcap} \;\; with \;\; M_{\sqcap} =_{df} M_{\sqcap}^{\mathfrak{S}^{\delta}} \lhd \mathfrak{S}^{\delta} \rhd M^{\neg \mathfrak{S}^{\delta}}$$

As this definition illustrates we need two merge relations for the quiescence preserving internal choice: $M_{\sqcap}^{\mathfrak{S}^{\delta}}$ and $M^{\neg \mathfrak{S}^{\delta}}$. $M_{\sqcap}^{\mathfrak{S}^{\delta}}$ merges the very beginning of the two processes P and Q. After that, $M^{\neg \mathfrak{S}^{\delta}}$ takes care that $P \sqcap^{\delta} Q$ behaves like P or Q.

$M_{\sqcap}^{\mathfrak{S}^{\delta}}$ and $M^{\neg \mathfrak{S}^{\delta}}$ share some common properties, formalized by M^{δ}: (1) Parts of P and Q are only merged if their $wait'$ values are equal, and (2) potentially the initial δ has to be removed from all traces.

Definition 4.11 (Internal choice - Common merge)

$$\begin{aligned}
M^{\delta} \quad =_{df} \quad & (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
& ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \\
& where\ initQuiet(t) =_{df} t \notin (\{s^\frown u | s \in \mathcal{A} \wedge u \in \mathcal{A}_{\delta}^{*}\} \cup \{\langle\rangle\})
\end{aligned}$$

The merge relation M^{δ} is symmetric in its variables, i.e. in $0.v = \{0.wait, 0.ok, 0.ref, 0.tr\}$ and in $1.v = \{1.wait, 1.ok, 1.ref, 1.tr\}$:

Lemma 4.23 (symmetric-M^{δ})

$$M^{\delta}[0.v, 1.v/1.v, 0.v] = M^{\delta}$$

Furthermore, the common merge relation M^{δ} is equivalent to having $wait' = (0.wait \wedge 1.wait)$ and $ok' = (0.ok \wedge 1.ok)$ in conjunction with the merge relation itself:

Lemma 4.24 (wait-and-ok-M^{δ})

$$M^{\delta} = (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M^{\delta}$$

$M_{\sqcap}^{\eth^{\delta}}$ can now be defined by the use of M^{δ}. In addition to the common merge relation, traces and refusal sets of P and Q are merged into new traces and new refusal sets.

Definition 4.12 (Internal choice - Initial merge)

$$\begin{array}{lll}
M_{\sqcap}^{\eth^{\delta}} & =_{df} & M^{\delta} \wedge M_{\sqcap}^{init} \\
M_{\sqcap}^{init} & =_{df} & ((tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))) \vee \\
 & & (tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))))
\end{array}$$

By adding $\{\delta\} \cap (0.ref \cup 1.ref)$ to the set of refused actions the new process refuses δ only if one of the two processes refuse to exhibit a δ event. In other words, only if both processes do not refuse δ, i.e., $\delta \notin 0.ref \wedge \delta \notin 1.ref$, the resulting process does not refuse δ as well, i.e., $\delta \notin ref'$.

The merge relation $M_{\sqcap}^{\eth^{\delta}}$ is symmetric in its variables, i.e. in $0.v = \{0.wait, 0.ok, 0.ref, 0.tr\}$ and in $1.v = \{1.wait, 1.ok, 1.ref, 1.tr\}$:

Lemma 4.25 (symmetric-$M_{\sqcap}^{\eth^{\delta}}$)

$$M_{\sqcap}^{\eth^{\delta}}[0.v, 1.v/1.v, 0.v] = M_{\sqcap}^{\eth^{\delta}}$$

Finally, we need to define $M^{\neg\eth^{\delta}}$. $M^{\neg\eth^{\delta}}$ takes care that finally $P \sqcap^{\delta} Q$ behaves like P or Q. Additionally, M^{δ} is applied in order to potentially remove δ from the traces.

Definition 4.13 (Internal choice - Terminal merge)

$$\begin{array}{lll}
M^{\neg\eth^{\delta}} & =_{df} & M^{\delta} \wedge M^{term} \\
M^{term} & =_{df} & ((tr' = 0.tr \wedge ref' = 0.ref) \vee (tr' = 1.tr \wedge ref' = 1.ref))
\end{array}$$

The merge relation $M^{\neg\eth^{\delta}}$ is symmetric in its variables, i.e. in $0.v = \{0.wait, 0.ok, 0.ref, 0.tr\}$ and in $1.v = \{1.wait, 1.ok, 1.ref, 1.tr\}$:

Lemma 4.26 (symmetric-$M^{\neg\eth^{\delta}}$)

$$M^{\neg\eth^{\delta}}[0.v, 1.v/1.v, 0.v] = M^{\neg\eth^{\delta}}$$

Our internal choice's merge relation is equal to setting the value of tr' to either $0.tr$ or $1.tr$ and the merge relation itself, i.e.,

Lemma 4.27 (tr-$M_\sqcap$)

$$M_\sqcap = (tr' = 0.tr \vee tr' = 1.tr) \wedge M_\sqcap$$

Furthermore, the merge relation $M_\sqcap$ is equivalent to having $wait' = (0.wait \wedge 1.wait)$ and $ok' = (0.ok \wedge 1.ok)$ in conjunction with the merge relation itself:

Lemma 4.28 (wait-and-ok-$M_\sqcap$)

$$M_\sqcap = (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M_\sqcap$$

We can also extract the calculation of the refusal sets from the merge relation:

Lemma 4.29 (wait-and-ref-$M_\sqcap$)

$$\begin{array}{l} M_\sqcap = M_\sqcap \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \\ \left(\left(\begin{array}{l} tr' = 0.tr \wedge ref'_{in} = 0.ref_{in} \wedge ref'_{out} = 0.ref_{out} \wedge \\ ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \vee \\ tr' = 1.tr \wedge ref'_{in} = 1.ref_{in} \wedge ref'_{out} = 1.ref_{out} \wedge \\ ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \end{array} \right) \lhd \mathfrak{F}^\delta \rhd \right. \\ \hfill \left. \left(\begin{array}{l} tr' = 0.tr \wedge ref' = 0.ref \wedge \\ ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge \\ ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{array} \right) \right) \end{array}$$

The merge relation $M_\sqcap$ is symmetric in its variables, i.e. in $0.v = \{0.wait, 0.ok, 0.ref, 0.tr\}$ and in $1.v = \{1.wait, 1.ok, 1.ref, 1.tr\}$:

Lemma 4.30 (symmetric-$M_\sqcap$)

$$M_\sqcap[0.v, 1.v/1.v, 0.v] = M_\sqcap$$

If the values for $0.v$ and $1.v$ in $M_\sqcap$ are the same, then $M_\sqcap$ does not change anything, i.e. it reduces to the relational skip of Cavalcanti and Woodcock (2004) $\mathbb{I}_{\text{rel}} =_{df} (wait' = wait) \wedge (ok' = ok) \wedge (ref' = ref) \wedge (tr' = tr)$.

Lemma 4.31 ($M_\sqcap$-reduces-to-skip)

$$M_\sqcap[v_0, v_0/1.v, 0.v] = \mathbb{I}_{rel}[v_0/v]$$

Our quiescence preserving internal choices $\sqcap^\delta$ obeys the usual laws for choices, i.e. quiescence preserving internal choices are an idempotent and commutative.

Lemma 4.32 (idempotent-$\sqcap^\delta$)

$$P \sqcap^\delta P = P$$

Lemma 4.33 (commutative-$\sqcap^\delta$)

$$P \sqcap^\delta Q = Q \sqcap^\delta P$$

Furthermore, our $\sqcap^\delta$ operator preserves healthiness with respect to **R1**, **R2**, **R3**$^\delta$, **IOCO1**, **IOCO2**, and **IOCO3**.

Lemma 4.34 (closure-$\sqcap^\delta$-R1)

$$\textbf{\textit{R1}}(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided P and Q are } \textbf{\textit{R1}} \textit{ healthy}$$

Lemma 4.35 (closure-$\sqcap^\delta$-R2)

$$\textbf{\textit{R2}}(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided P and Q are } \textbf{\textit{R2}} \textit{ healthy}$$

Lemma 4.36 (closure-$\sqcap^\delta$-R3$^\delta$)

$$\textbf{\textit{R3}}^\delta(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided P and Q are } \textbf{\textit{R3}}^\delta \textit{ healthy}$$

Lemma 4.37 (closure-$\sqcap^\delta$-IOCO1)

$$\textbf{\textit{IOCO1}}(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided P and Q are } \textbf{\textit{IOCO1}} \textit{ healthy}$$

Lemma 4.38 (closure-$\sqcap^\delta$-IOCO2)

$$\textbf{\textit{IOCO2}}(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided P and Q are } \textbf{\textit{IOCO2}} \textit{ healthy}$$

Lemma 4.39 (closure-$\sqcap^\delta$-IOCO3)

$$\textbf{\textit{IOCO3}}(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided P and Q are } \textbf{\textit{IOCO3}} \textit{ healthy}$$

A quiescence preserving external choice operator is given by

Definition 4.14 (Quiescence preserving external choice)

$$P +^\delta Q \quad =_{df} \quad (P \curlywedge Q); M_+ \;\; with \;\; M_+ =_{df} M_+^{\eth^\delta} \lhd \eth^\delta \rhd M^{\neg\eth^\delta}$$

Our external choice's merge relation is equal to setting the value of tr' to either $0.tr$ or $1.tr$ and the merge relation itself, i.e.,

Lemma 4.40 (tr-M_+)

$$M_+ = (tr' = 0.tr \vee tr' = 1.tr) \wedge M_+$$

Furthermore, the merge relation M_+ is equivalent to having $wait' = (0.wait \wedge 1.wait)$ and $ok' = (0.ok \wedge 1.ok)$ in conjunction with the merge relation itself:

Lemma 4.41 (wait-and-ok-M_+)

$$M_+ = (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M_+$$

As for the internal choice initial merge relation we can extract the calculation of the refusals for M_+ as well.

Lemma 4.42 (wait-and-ref-M_+)

$$\begin{array}{l} M_+ = M_+ \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \\ \left(\left(\begin{array}{l} tr' = 0.tr \wedge ref'_{in} = 0.ref_{in} \wedge ref'_{out} = 0.ref_{out} \wedge \\ ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \vee \\ tr' = 1.tr \wedge ref'_{in} = 1.ref_{in} \wedge ref'_{out} = 1.ref_{out} \wedge \\ ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \end{array} \right) \lhd \eth^\delta \rhd \right. \\ \hfill \left. \left(\begin{array}{l} tr' = 0.tr \wedge ref' = 0.ref \wedge \\ ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge \\ ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{array} \right) \right) \end{array}$$

The merge relation M_+ is symmetric in its variables, i.e. in $0.v = \{0.wait, 0.ok, 0.ref, 0.tr\}$ and in $1.v = \{1.wait, 1.ok, 1.ref, 1.tr\}$:

Lemma 4.43 (symmetric-M_+)

$$M_+[0.v, 1.v/1.v, 0.v] = M_+$$

Except for the merge relation $M_+^{\circledS\delta}$ the external choice is equivalent $\sqcap^\delta$. The difference is how the very beginning of P and Q is combined to form $P +^\delta Q$.

Definition 4.15 (External choice merge relation)

$$\begin{array}{rcl} M_+^{\circledS\delta} & =_{df} & M^\delta \wedge M_+^{init} \\ M_+^{init} & =_{df} & (ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))) \wedge \\ & & (tr' = 0.tr \vee tr' = 1.tr) \end{array}$$

The merge relation $M_+^{\circledS\delta}$ is symmetric in its variables, i.e. in $0.v = \{0.wait, 0.ok, 0.ref, 0.tr\}$ and in $1.v = \{1.wait, 1.ok, 1.ref, 1.tr\}$:

Lemma 4.44 (symmetric-$M_+^{\circledS\delta}$)

$$M_+^{\circledS\delta}[0.v, 1.v/1.v, 0.v] = M_+^{\circledS\delta}$$

As $M_\sqcap$, M_+ does not change anything if the values for $0.v$ and $1.v$ in $M_\sqcap$ are the same, i.e. it reduces to the relational skip $\mathbb{I}_{\text{rel}}$.

Lemma 4.45 (M_+-reduces-to-skip)

$$M_+[v_0, v_0/1.v, 0.v] = \mathbb{I}_{rel}[v_0/v]$$

Also, our quiescence preserving external choices $+^\delta$ obeys the laws for quiescent preserving internal choices.

Lemma 4.46 (idempotent-$+^\delta$)

$$P +^\delta P = P$$

Lemma 4.47 (commutative-$+^\delta$)

$$P +^\delta Q = Q +^\delta P$$

Our healthiness conditions are preserved by $+^\delta$.

Lemma 4.48 (closure-$+^\delta$-R1)

$$\boldsymbol{R1}(P +^\delta Q) = P +^\delta Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{R1} \textit{ healthy}$$

Lemma 4.49 (closure-$+^\delta$-R2)

$$\boldsymbol{R2}(P +^\delta Q) = P +^\delta Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{R2} \textit{ healthy}$$

Lemma 4.50 (closure-$+^\delta$-R3$^\delta$)

$$\boldsymbol{R3}^\delta(P +^\delta Q) = P +^\delta Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{R3}^\delta \textit{ healthy}$$

Lemma 4.51 (closure-$+^\delta$-IOCO1)

$$\boldsymbol{IOCO1}(P +^\delta Q) = P +^\delta Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO1} \textit{ healthy}$$

Lemma 4.52 (closure-$+^\delta$-IOCO2)

$$\boldsymbol{IOCO2}(P +^\delta Q) = P +^\delta Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO2} \textit{ healthy}$$

Lemma 4.53 (closure-$+^\delta$-IOCO3)

$$\boldsymbol{IOCO3}(P +^\delta Q) = P +^\delta Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO3} \textit{ healthy}$$

Specification processes for the **ioco** framework are defined as follows:

Definition 4.16 (ioco specification) *A* **ioco** *specification is a reactive process satisfying the healthiness conditions* ***IOCO1****,* ***IOCO2*** *and* ***IOCO3****. In addition its set of possible events is partitioned into the quiescent event, input events, and output events: $\mathcal{A} = \mathcal{A}_{out} \cup \mathcal{A}_{in} \cup \{\delta\}$ where $\mathcal{A}_{out} \cap \mathcal{A}_{in} = \emptyset$ and $\delta \notin \mathcal{A}_{out} \cup \mathcal{A}_{in}$*

Processes expressed in terms of $do^\delta_\mathcal{A}$, ;, $+^\delta$ and $\sqcap^\delta$ are **ioco** specifications.

Theorem 4.1 *The set of* **ioco** *specifications is a $\{;, +^\delta, \sqcap^\delta\}$-closure.*

Proof 4.1. Idempotence of healthiness conditions and healthiness conditions are preserved (see lemmas)□

Remark 4.1 *The class of labeled transition systems (LTS) used for the* **ioco** *relation is restricted to image finite LTSs (Tretmans, 2008). Image finite LTSs are limited in their possible non-deterministic choices, i.e. image finite LTSs are bounded in terms of non-determinism. This requirement is only due to the properties of Tretmans' test case generation algorithm. Since we are interested in a predicative semantics we do not face the problem of image-finiteness.* □

Remark 4.2 *The* TGV *tool (Jard and Jéron, 2005), which claims to generate test cases with respect to* **ioco***, uses a different notion of quiesence: $quiesence_{TGV} =_{df} quiesence \vee (\neg ok' \wedge \neg wait')$. Note that we rely on quiescence rather than on $quiescence_{TGV}$.* □

4.3 IOCO Implementations

The input-output conformance relation uses labeled transition systems to represent implementations. As mentioned in Section 2.3, it is not assumed that this LTS is known in advance, but only its existence is required. Our formalization requires something similar: implementations can be expressed as processes.

Processes for representing implementations in terms of the **ioco** relation need to satisfy the properties of specifications plus three additional properties: some restrictions on allowed choices, input-enabledness, and fairness.

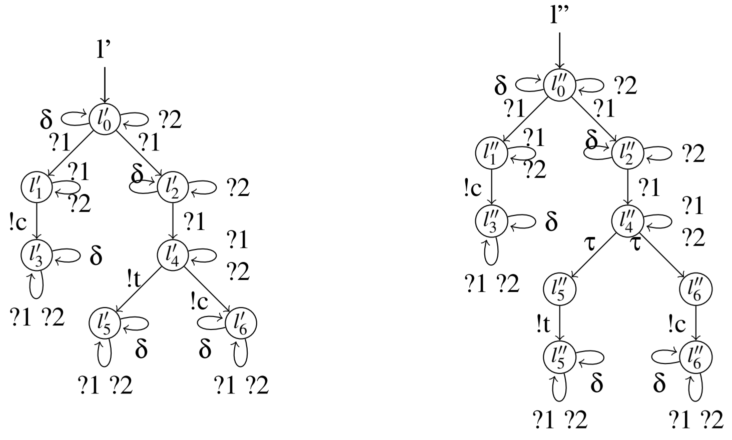


Figure 4.1: **IOCO4** causes choices between outputs to be internal choices.

Restrictions on choices.

An implementation is not allowed to freely choose between the actions enabled in a particular state. The **ioco** relation distinguishes between inputs and outputs not only by partitioning a process's alphabet, but also by assigning responsibilities to these two alphabets. According to Tretmans (1996) "Outputs are actions that are initiated by and under control of an implementation under test, while input actions are initiated by and under control of the system's environment".

In terms of choices this means that for implementations choices between outputs are internal choices. Internal choices are represented by disjunctions over the refused actions, thus we restrict the choices between outputs:

IOCO4 $\qquad P = P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$

Example 4.3. The effect of this healthiness condition is illustrated by Figure 4.1, i.e., the choice between $!t$ and $!c$ in state l'_4 is an internal choice. Thus, as required by **IOCO4**, after the trace $\langle ?1, \delta, ?1 \rangle$ l'' does not offer $!t$ and $!c$ for communication, i.e. $!t \notin ref' \land !c \notin ref'$. Instead, it non-deterministically offers only one of the two actions for communication, i.e. $!t \notin ref' \lor !c \notin ref'$. □

On the contrary, input actions are under control of the system's environment. That is, choices between inputs are external choices. This restriction is enforced by requiring input-enabledness (see **IOCO5**).

In addition, if there are inputs and outputs enabled in a particular state of an implementation the choice between input and output is up to the environment. That is, choices between inputs and outputs are external choices. Again, this restriction is covered by having input-enabled implementations, i.e. $ref'_{in} = \emptyset$ (see **IOCO5**).

Remark 4.3 *Recently, the constraints on the semantics of choices within implementations have been relaxed (Tretmans, 2008). By making test cases input enabled, choices between inputs and outputs are now choices of the implementation. However, our work focuses on the original definition of* **ioco**. □

As a function **IOCO4** can be defined as $\mathbf{IOCO4}(P) = P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$. This function is an idempotent.

Lemma 4.54 (IOCO4-idempotent)

$$\mathbf{IOCO4} \circ \mathbf{IOCO4} = \mathbf{IOCO4}$$

Furthermore, **IOCO4** commutes with our other healthiness conditions.

Lemma 4.55 (commutativity-IOCO4-R1)

$$\boldsymbol{IOCO4} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{IOCO4}$$

Lemma 4.56 (commutativity-IOCO4-R2)

$$\boldsymbol{IOCO4} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{IOCO4}$$

Lemma 4.57 (commutativity-IOCO4-R3)

$$\boldsymbol{IOCO4} \circ \boldsymbol{R3} = \boldsymbol{R3} \circ \boldsymbol{IOCO4}$$

Lemma 4.58 (commutativity-IOCO4-IOCO1)

$$\boldsymbol{IOCO4} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{IOCO4}$$

Lemma 4.59 (commutativity-IOCO4-IOCO2)

$$\boldsymbol{IOCO4} \circ \boldsymbol{IOCO2} = \boldsymbol{IOCO2} \circ \boldsymbol{IOCO4}$$

Lemma 4.60 (commutativity-IOCO4-IOCO3)

$$\boldsymbol{IOCO4} \circ \boldsymbol{IOCO3} = \boldsymbol{IOCO3} \circ \boldsymbol{IOCO4}$$

IOCO4 is preserved by sequential composition (;), by quiescence preserving internal choice ($\sqcap^\delta$), and by quiescence preserving external choice ($+^\delta$):

Lemma 4.61 (closure-;-IOCO4)

$$\boldsymbol{IOCO4}(P;Q) = P;Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO4} \textit{ healthy and } Q \textit{ is } \boldsymbol{R3}^\delta_\iota \textit{ healthy}$$

Lemma 4.62 (closure-$\sqcap^\delta$-IOCO4)

$$\boldsymbol{IOCO4}(P \sqcap^\delta Q) = P \sqcap^\delta Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO4} \textit{ healthy}$$

Lemma 4.63 (closure-$+^\delta$-IOCO4)

$$\boldsymbol{IOCO4}(P +^\delta Q) = P \sqcap^\delta Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO4} \textit{ healthy}$$

Input-enabledness.

Input-enabledness requires that an implementation accepts every input in every (waiting) state. More precisely, an implementation cannot prevent the environment from providing an input, while running.

IOCO5 $\quad P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$

As a function **IOCO5** can be defined as $\mathbf{IOCO5}(P) = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$. This function is an idempotent.

Lemma 4.64 (IOCO5-idempotent)

$$\boldsymbol{IOCO5} \circ \boldsymbol{IOCO5} = \boldsymbol{IOCO5}$$

Furthermore, **IOCO5** commutes with our other healthiness conditions.

Lemma 4.65 (commutativity-IOCO5-R1)

$$\boldsymbol{IOCO5} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{IOCO5}$$

Lemma 4.66 (commutativity-IOCO5-R2)

$$\boldsymbol{IOCO5} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{IOCO5}$$

Lemma 4.67 (commutativity-IOCO5-R3)

$$\boldsymbol{IOCO5} \circ \boldsymbol{R3} = \boldsymbol{R3} \circ \boldsymbol{IOCO5}$$

Lemma 4.68 (commutativity-IOCO5-IOCO1)

$$\boldsymbol{IOCO5} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{IOCO5}$$

Lemma 4.69 (commutativity-IOCO5-IOCO2)

$$\boldsymbol{IOCO5} \circ \boldsymbol{IOCO2} = \boldsymbol{IOCO2} \circ \boldsymbol{IOCO5}$$

Lemma 4.70 (commutativity-IOCO5-IOCO3)

$$\boldsymbol{IOCO5} \circ \boldsymbol{IOCO3} = \boldsymbol{IOCO3} \circ \boldsymbol{IOCO5}$$

As for specifications we need to redefine $\mathbb{I}$ such that even in the case of divergence the additional properties of implementations are satisfied.

$$\mathbb{I}^{\delta}_{\iota} =_{df} \left(\begin{array}{l} \neg ok \wedge (tr \le tr') \wedge (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))\wedge \\ (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s ^\frown \delta^*))\wedge \\ (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \le |ref'_{out}|) \wedge (wait' \Rightarrow (ref'_{in} = \emptyset)) \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \\ (v' = v) \end{array} \right)$$

Using this new version of the $\mathbb{I}$-relation within the healthiness condition **R3** leads to $\mathbf{R3}^{\delta}_{\iota}$ which is used for implementations

$$\mathbf{R3}^{\delta}_{\iota}(P) =_{df} \mathbb{I}^{\delta}_{\iota} \lhd wait \rhd P$$

For $\mathbf{R3}^{\delta}_{\iota}$-healthy processes, **IOCO5** is preserved by sequential composition (;), by quiescence preserving internal choice ($\sqcap^{\delta}$), and by quiescence preserving external choice ($+^{\delta}$):

Lemma 4.71 (closure-;-IOCO5)

$$\boldsymbol{IOCO5}(P;Q) = P;Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO5} \textit{ healthy and } Q \textit{ is } \boldsymbol{R3}^{\delta}_{\iota} \textit{ healthy}$$

Lemma 4.72 (closure-$\sqcap^{\delta}$-IOCO5)

$$\boldsymbol{IOCO5}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO5} \textit{ healthy}$$

Lemma 4.73 (closure-$+^{\delta}$-IOCO5)

$$\boldsymbol{IOCO5}(P +^{\delta} Q) = P \sqcap^{\delta} Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO5} \textit{ healthy}$$

While specifications are expressed in terms of $do^{\delta}_{\mathcal{A}}$, implementations use $\iota_{\mathcal{A}}$. More precisely, we express implementations by the use of $\iota^{\delta}_{\mathcal{A}}$. But let us start with $\iota_{\mathcal{A}}$ first. $\iota_{\mathcal{A}}$ takes care of the input-enabledness of processes.

For the sake of simplicity we use the following abbreviation to denote a sequence of inputs without a particular action: $\mathcal{A}^{*}_{in\backslash a} =_{df} (\mathcal{A}_{in} \setminus \{a\})^{*}$.

Definition 4.17 (Input-enabled communication) *Let $a \in \mathcal{A}$ be an action of a process's alphabet, then*

$$\iota_{\mathcal{A}}(a) =_{df} \Phi^{i}(ref'_{in} = \emptyset \wedge a \notin ref' \wedge tr' - tr \in \mathcal{A}^{*}_{in\backslash a} \lhd wait' \rhd tr' - tr \in \mathcal{A}^{*}_{in\backslash a} {}^\frown \langle a \rangle)$$

$\iota_{\mathcal{A}}(a)$ is similar to $do_{\mathcal{A}}(a)$. It denotes that the process $\iota_{\mathcal{A}}(a)$ cannot refuse to perform an a-action. Furthermore, $\iota_{\mathcal{A}}(a)$ cannot refuse to perform any input action. After executing any input action sequence ended by an a action the process $\iota_{\mathcal{A}}(a)$ terminates successfully.

Input enabledness also affects the representation of a deadlock. An input-enabled process needs to accept an input action at any time. That is an input-enabled process can only deadlock on outputs. Therefore, the deadlock process δ_{ι}, which substitutes δ in the case of input-enabled processes, is given by:

Definition 4.18 (Output deadlock)

$$\delta_{\iota} =_{df} \boldsymbol{R3}^{\delta}_{\iota}(tr' - tr \in \mathcal{A}^{*}_{in} \wedge wait')$$

The output deadlock is a left-zero for sequential composition

Lemma 4.74 (δ_{ι}-left-zero)

$$\delta_{\iota}; P = \delta_{\iota}$$

Again, as for the non-input-enabled case, we need a quiescent version of $\iota_{\mathcal{A}}$. $\iota^{\delta}_{\mathcal{A}}(a)$ has δ events within its traces if a is an input event.

Definition 4.19 (Input-enabled quiescent communication) *Let $a \in \mathcal{A}$ be an action of a process's alphabet, then*

$$\iota^{\delta}_{\mathcal{A}}(a) =_{df} \begin{cases} \iota_{\mathcal{A}}(a) & \textit{if } a \in \mathcal{A}_{out} \\ \Phi^{i}\left(\left(\begin{array}{l} ref'_{in} = \emptyset \wedge \delta \notin ref' \wedge \\ tr' - tr \in (\mathcal{A}_{in\backslash a} \cup \delta)^{*} \end{array}\right) \lhd wait' \rhd tr' - tr \in (\mathcal{A}_{in\backslash a} \cup \delta)^{*} {}^\frown \langle a \rangle\right) & \textit{if } a \in \mathcal{A}_{in} \end{cases}$$

Combining input-enabledness with quiescence again requires a slight modification of the deadlock process. This leads to the quiescent output deadlock:

Definition 4.20 (Quiescent output deadlock)

$$\delta^{\delta}_{\iota} =_{df} \boldsymbol{R3}^{\delta}_{\iota}(tr' - tr \in (\mathcal{A}_{in} \cup \delta)^{*} \wedge wait')$$

The quiescent output deadlock is a left-zero for sequential composition

Lemma 4.75 (δ^{δ}_{ι}-left-zero)

$$\delta^{\delta}_{\iota}; P = \delta^{\delta}_{\iota}$$

Fairness.

Fairness is especially important for allowing theoretical exhaustive test case generation algorithms. The fairness assumption for the **ioco** relation requires that an implementation eventually shows all its possible non-deterministic behaviors when it is re-executed with a particular set of inputs. Without assuming fairness of implementations there is not even a theoretical possibility of generating a failing test case for any non-conforming implementation. An unfair implementation may always lead a test case away from its errors. To express this fairness on an implementation we use a probabilistic choice operator similar to He and Sanders (2006). According to He and Sanders (2006) a probabilistic choice between two processes A and B is expressed by $A\ {}_{p}\oplus\ B$ where $0 \leq p \leq 1$. This expression equals A with probability p and B with probability $1-p$. For example, $A\ {}_{0.9}\oplus\ B$ denotes that during execution A is chosen in 90% of the cases.

The probabilistic version of our quiescence preserving internal choice, i.e. $\sqcap^{\delta}$, is given by ${}_{p}\textcircled{$\sqcap$}^{\delta}$. The laws for ${}_{p}\textcircled{$\sqcap$}^{\delta}$ are similar to the laws for ${}_{p}\oplus$, i.e.,

$$\begin{array}{rcll}
P\ {}_{1}\textcircled{$\sqcap$}^{\delta}\ Q & = & P & \\
P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q & = & Q\ {}_{1-p}\textcircled{$\sqcap$}^{\delta}\ P & \\
P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ P & = & P & \\
(P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q)\ {}_{q}\textcircled{$\sqcap$}^{\delta}\ R & = & P\ {}_{pq}\textcircled{$\sqcap$}^{\delta}\ (Q\ {}_{r}\textcircled{$\sqcap$}^{\delta}\ R), & \qquad r = ((1-p)q)/(1-(pq)) \\
(P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q);R & = & (P;R)\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ (Q;R)) &
\end{array}$$

A quiescence preserving internal choice is given by the non-deterministic choice of all possible probabilistic choices, i.e.

$$P\ \sqcap^{\delta}\ Q = \sqcap\{P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q | 0 \leq p \leq 1\} \sqsubseteq P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q$$

Relying on probabilistic choices means that when one implements a choice the specification is refined by choosing a particular probability for this choice. Fairness is expressed by restricting the probabilities p to $0 < p < 1$:

Definition 4.21 (Internal fair (quiescence preserving) choice)

$$P \sqcap^{\delta}_{f} Q \quad =_{df} \quad \sqcap\{P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q | 0 < p < 1\}$$

Thus, when executing a test case on the implementation the implementation will eventually exhibit all its possible behavior. As stated by the following lemma fair internal quiescence preserving choices are valid implementations of internal quiescence preserving choices. This guarantees that our internal quiescence preserving choice can be safely implemented by its fair version.

Lemma 4.76 (Fair choice refinement)

$$P\ \sqcap^{\delta}\ Q \sqsubseteq P \sqcap^{\delta}_{f} Q$$

Given the notion of input-enabledness and fairness we can now define which processes serve to represent **ioco** testable implementations:

Definition 4.22 (ioco testable implementation) *A* **ioco** *testable implementation is a reactive process satisfying the healthiness conditions **IOCO1-IOCO5**. In addition, a* **ioco** *testable implementation must be fair.*

Processes expressed in terms of $\iota^{\delta}_{\mathcal{A}}$, ;, $+^{\delta}$, and $\sqcap^{\delta}_{f}$ are **ioco** testable implementations if their choices obey the following rules: (1) Choices between outputs are fair internal choices ($\sqcap^{\delta}_{f}$); (2) Choices between inputs and choices between inputs and outputs are external choices ($+^{\delta}$).

Theorem 4.2 *Implementation processes are closed under* $\iota^{\delta}_{\mathcal{A}}$, ;, $+^{\delta}$, *and* $\sqcap^{\delta}_{f}$.

Proof 4.2. Idempotence of healthiness conditions and healthiness conditions are preserved (see lemmas)□

4.4 Predicative Input-Output Conformance Relation

Recall that informally an IUT conforms to a specification S, iff the outputs of the IUT are outputs of S after an arbitrary suspension trace of S. Thus, we need the (suspension) traces of a process, which are obtained by hiding all observations except the traces.

Definition 4.23 (Traces of a process)

$$Trace(P) =_{df} \exists ref, ref', wait, wait', ok, ok' \bullet P$$

In addition to all traces of a particular process we need the traces after which a process is quiescent. Due to the chosen representation of quiescence, i.e. $quiescence = \neg wait' \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref'$ (see Section 4.2), we use the following predicate in order to obtain the traces after which a process is quiescent.

Definition 4.24 (Quiet traces of a process)

$$Quiet(P) =_{df} \exists ref'_{in} \bullet (P[false/wait'] \vee P[A_{out}/ref'_{out}])$$

Using these two predicates the input-output conformance relation between implementation processes (see Definition 4.22) and specification processes (see Definition 4.16) is defined as follows:

Definition 4.25 ($\sqsubseteq_{ioco}$) *Given an implementation process I and a specification process S, then*

$$\begin{aligned} S \sqsubseteq_{ioco} I =_{df} [\, \forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet & ((Trace(S)[t/tr'] \wedge Trace(I)[t^\frown o/tr']) \Rightarrow Trace(S)[t^\frown o/tr']) \wedge \\ & ((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr'])\,] \end{aligned}$$

In order to distinguish the input-output conformance given in denotational semantics from its operational semantics version we use different symbols. Note that because $\sqsubseteq_{ioco}$ is related to refinement I **ioco** S is given by $S \sqsubseteq_{ioco} I$.

ioco relates the outputs (including quiescence) of I and S for all suspension traces of S. On the contrary, our $\sqsubseteq_{ioco}$ definition comprises two different parts. The first part considers only outputs while the second part deals with quiescence.

Although, $\sqsubseteq_{ioco}$ and **ioco** are not transitive in general, an interesting property is that refining a conforming implementation does not break conformance.

Theorem 4.3 $((S \sqsubseteq_{ioco} I_2) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow S \sqsubseteq_{ioco} I_1$

Proof 4.3.

$$\begin{aligned}
& S \sqsubseteq_{ioco} I_2 \wedge I_2 \sqsubseteq I_1 \\
& \qquad \text{(definition of } \sqsubseteq \text{ and propositional calculus)} \\
= \; & (S \sqsubseteq_{ioco} I_2) \wedge [I_1 \Rightarrow I_2] \wedge [I_1 \Rightarrow I_2] \\
& \qquad \text{(def. of } \exists, \, Trace, \text{ and } Quiet) \\
\Rightarrow & (S \sqsubseteq_{ioco} I_2) \wedge [Trace(I_1) \Rightarrow Trace(I_2)] \wedge [Quiet(I_1) \Rightarrow Quiet(I_2)] \\
& \qquad \text{(propositional calculus)} \\
\Rightarrow & [\, \forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet (((Trace(S)[t/tr'] \wedge Trace(I_2)[t^\frown o/tr']) \Rightarrow Trace(S)[t^\frown o/tr']) \wedge \\
& \qquad (Trace(I_1)[t^\frown o/tr'] \Rightarrow Trace(I_2)[t^\frown o/tr'])) \wedge \\
& \quad \forall t \in \mathcal{A}_\delta^* \bullet (\, ((Trace(S)[t/tr'] \wedge Quiet(I_2)[t/tr']) \Rightarrow Quiet(S)[t/tr']) \wedge \\
& \qquad (Quiet(I_1)[t/tr'] \Rightarrow Quiet(I_2)[t/tr']))] \\
& \qquad \text{(propositional calculus)} \\
\Rightarrow & [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet (((Trace(S)[t/tr'] \wedge Trace(I_1)[t^\frown o/tr']) \Rightarrow Trace(S)[t^\frown o/tr']) \wedge \\
& \qquad ((Trace(S)[t/tr'] \wedge Quiet(I_1)[t/tr']) \Rightarrow Quiet(S)[t/tr']))] \\
& \qquad \text{(def. of } \sqsubseteq_{ioco}) \\
= \; & S \sqsubseteq_{ioco} I_1 \qquad \Box
\end{aligned}$$

Lemma 4.77 (not-ioco and refinement)

$$(((S \sqsubseteq_{ioco} I_2) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow S \sqsubseteq_{ioco} I_1) = (((S \not\sqsubseteq_{ioco} I_1) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow S \not\sqsubseteq_{ioco} I_2)$$

Example 4.4. The denotational version of *i* **ioco** *e* (as shown in Example 4) can be calculated by using the processes $\mathcal{P}_i = \iota^{\delta}_{\mathcal{A}}(?1);\iota^{\delta}_{\mathcal{A}}(!c)$ and $\mathcal{P}_e = do^{\delta}_{\mathcal{A}}(?1);do^{\delta}_{\mathcal{A}}(!c)$ as follows:

$\mathcal{P}_e \sqsubseteq_{ioco} \mathcal{P}_i =$

(definition of $\sqsubseteq_{ioco}$)

$=[\forall t \in \mathcal{A}^*_{\delta}, \forall o \in \mathcal{A}_{out} \bullet ((Trace(\mathcal{P}_e)[t/tr'] \wedge Trace(\mathcal{P}_i)[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_e)[t^\frown o/tr']) \wedge$
$((Trace(\mathcal{P}_e)[t/tr'] \wedge Quiet(\mathcal{P}_i)[t/tr']) \Rightarrow Quiet(\mathcal{P}_e)[t/tr'])]$

(definition of $\mathcal{P}_e$, of $\mathcal{P}_i$, and of $Trace$)

$=[\forall t \in \mathcal{A}^*_{\delta}, \forall o \in \mathcal{A}_{out} \bullet (((t \in tr^\frown\delta^* \vee t \in tr^\frown\delta^{*\frown}\langle ?1\rangle \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle) \wedge$
$(t^\frown o \in tr^\frown(\mathcal{A}_{in\setminus ?1} \cup \delta)^* \vee t^\frown o \in tr^\frown(\mathcal{A}_{in} \cup \delta)^{*\frown}\langle ?1\rangle \vee$
$t^\frown o \in tr^\frown(\mathcal{A}_{in\setminus ?1} \cup \delta)^{*\frown}\langle ?1\rangle^\frown\mathcal{A}^{*\frown}_{in}\langle !c\rangle)) \Rightarrow$
$(t^\frown o \in tr^\frown\delta^* \vee t^\frown o \in tr^\frown\delta^{*\frown}\langle ?1\rangle \vee t^\frown o \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle)) \wedge$
$((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]$

(propositional calculus)

$=[\forall t \in \mathcal{A}^*_{\delta} \bullet ((\neg(t^\frown\langle !c\rangle \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle) \vee t^\frown\langle !c\rangle \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle) \wedge$
$((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]$

(definition of *Quiet* and definition of *Trace*)

$=[\forall t \in \mathcal{A}^*_{\delta} \bullet (((t \in tr^\frown\delta^* \vee t \in tr^\frown\delta^{*\frown}\langle ?1\rangle \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle) \wedge$
$(t \in tr^\frown(\mathcal{A}_{in\setminus ?1} \cup \delta)^* \vee t \in tr^\frown(\mathcal{A}^*_{in\setminus ?1} \cup \delta)^{*\frown}\langle ?1\rangle^\frown\mathcal{A}^{*\frown}_{in}\langle !c\rangle)) \Rightarrow$
$(t \in tr^\frown\delta^* \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle))]$

(propositional calculus)

$=[\forall t \in \mathcal{A}^*_{\delta} \bullet TRUE]$

(definition of $\forall$ and definition of [])

$=TRUE$

□

Example 4.5. An implementation is allowed to provide fewer outputs than the specification. This is illustrated by the following example. Let $\mathcal{P}_{spec} = do^{\delta}_{\mathcal{A}}(?1);(do^{\delta}_{\mathcal{A}}(!c) \sqcap^{\delta} do^{\delta}_{\mathcal{A}}(!t))$ and let $\mathcal{P}_{iut} = \iota^{\delta}_{\mathcal{A}}(?1);\iota^{\delta}_{\mathcal{A}}(!c)$, then

$\mathcal{P}_{spec} \sqsubseteq_{ioco} \mathcal{P}_{iut} =$

(definition of $\sqsubseteq_{ioco}$)

$=[\forall t \in \mathcal{A}^*_{\delta}, \forall o \in \mathcal{A}_{out} \bullet ((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Trace(\mathcal{P}_{iut})[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_{spec})[t^\frown o/tr']) \wedge$
$((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Quiet(\mathcal{P}_{iut})[t/tr']) \Rightarrow Quiet(\mathcal{P}_{spec})[t/tr'])]$

(definition of $\mathcal{P}_{spec}$, of $\mathcal{P}_{iut}$, and of $Trace$)

$=[\forall t \in \mathcal{A}^*_{\delta}, \forall o \in \mathcal{A}_{out} \bullet (((t \in tr^\frown\delta^* \vee t \in tr^\frown\delta^{*\frown}\langle ?1\rangle \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !t\rangle) \wedge$
$(t^\frown o \in tr^\frown(\mathcal{A}_{in\setminus ?1} \cup \delta)^* \vee t^\frown o \in tr^\frown(\mathcal{A}_{in} \cup \delta)^{*\frown}\langle ?1\rangle \vee$
$t^\frown o \in tr^\frown(\mathcal{A}_{in\setminus ?1} \cup \delta)^{*\frown}\langle ?1\rangle^\frown\mathcal{A}^{*\frown}_{in}\langle !c\rangle)) \Rightarrow$
$(t^\frown o \in tr^\frown\delta^* \vee t^\frown o \in tr^\frown\delta^{*\frown}\langle ?1\rangle \vee t^\frown o \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle \vee$
$t^\frown o \in tr^\frown\delta^{*\frown}\langle ?1, !t\rangle)) \wedge$
$((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]$

(propositional calculus)

$=[\forall t \in \mathcal{A}^*_{\delta} \bullet ((\neg(t^\frown\langle !c\rangle \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle) \vee t^\frown\langle !c\rangle \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle) \wedge$
$((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]$

(definition of *Quiet* and definition of *Trace*)

$=[\forall t \in \mathcal{A}^*_{\delta} \bullet TRUE \wedge$
$(((t \in tr^\frown\delta^* \vee t \in tr^\frown\delta^{*\frown}\langle ?1\rangle \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !c\rangle \vee t \in tr^\frown\delta^{*\frown}\langle ?1, !t\rangle) \wedge$
$(t \in tr^\frown(\mathcal{A}_{in\setminus ?1} \cup \delta)^* \vee t \in tr^\frown(\mathcal{A}^*_{in\setminus ?1} \cup \delta)^{*\frown}\langle ?1\rangle^\frown\mathcal{A}^{*\frown}_{in}\langle !c\rangle)) \Rightarrow$

$$(t \in tr^\frown \delta^* \vee t \in tr^\frown \delta^{*\frown} \langle ?1, !c\rangle \ \vee\ t \in tr^\frown \delta^{*\frown} \langle ?1, !t\rangle))]$$

(propositional calculus)

$$= [\forall t \in \mathcal{A}_\delta^* \bullet TRUE \wedge TRUE]$$

(definition of $\forall$ and definition of $[]$)

$$= TRUE$$

$\square$

Example 4.6. If an implementation provides an output that is not allowed by the specification we have a non-conforming implementation. For example, let $\mathcal{P}_{spec} = do_{\mathcal{A}}^\delta(?1); do_{\mathcal{A}}^\delta(!c)$ be the specification and let $\mathcal{P}_{iut} = \iota_{\mathcal{A}}^\delta(?1); \iota_{\mathcal{A}}^\delta(!c) \sqcap_f^\delta \iota_{\mathcal{A}}^\delta(!t)$ be the implementation process, then we have $\neg(\mathcal{P}_{spec} \sqsubseteq_{ioco} \mathcal{P}_{iut})$. Note that for this example we need $\mathcal{P}'_{iut}$ which is given by $\mathcal{P}'_{iut} = \iota_{\mathcal{A}}^\delta(?1); \iota_{\mathcal{A}}^\delta(!c) \sqcap^\delta \iota_{\mathcal{A}}^\delta(!t)$

$$\mathcal{P}_{spec} \sqsubseteq_{ioco} \mathcal{P}'_{iut}$$

(definition of $\sqsubseteq_{ioco}$)

$$\begin{aligned}
= [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet & ((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Trace(\mathcal{P}'_{iut})[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_{spec})[t^\frown o/tr']) \wedge \\
& ((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Quiet(\mathcal{P}'_{iut})[t/tr']) \Rightarrow Quiet(\mathcal{P}_{spec})[t/tr'])]
\end{aligned}$$

(definition of $\mathcal{P}_{spec}$, of $\mathcal{P}_{iut}$, and of $Trace$)

$$\begin{aligned}
= [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet & (((t \in tr^\frown \delta^* \ \vee\ t \in tr^\frown \delta^{*\frown} \langle ?1\rangle \ \vee\ t \in tr^\frown \delta^{*\frown} \langle ?1, !c\rangle) \wedge \\
& (t^\frown o \in tr^\frown (\mathcal{A}_{in \setminus ?1} \cup \delta)^* \ \vee\ t^\frown o \in tr^\frown (\mathcal{A}_{in} \cup \delta)^{*\frown} \langle ?1\rangle \ \vee \\
& \ t^\frown o \in tr^\frown (\mathcal{A}_{in \setminus ?1} \cup \delta)^{*\frown} \langle ?1\rangle^\frown \mathcal{A}_{in}^{*\frown} \langle !c\rangle \ \vee \\
& \ t^\frown o \in tr^\frown (\mathcal{A}_{in \setminus ?1} \cup \delta)^{*\frown} \langle ?1\rangle^\frown \mathcal{A}_{in}^{*\frown} \langle !t\rangle)) \Rightarrow \\
& (t^\frown o \in tr^\frown \delta^* \ \vee\ t^\frown o \in tr^\frown \delta^{*\frown} \langle ?1\rangle \ \vee\ t^\frown o \in tr^\frown \delta^{*\frown} \langle ?1, !c\rangle)) \wedge \\
& ((Trace(S)[t/tr'] \ \wedge\ Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]
\end{aligned}$$

(propositional calculus)

$$\begin{aligned}
= [\forall t \in \mathcal{A}_\delta^* \bullet & ((\neg(t^\frown \langle !t\rangle \in tr^\frown \delta^{*\frown} \langle ?1, !t\rangle)) \wedge \\
& ((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]
\end{aligned}$$

(distributivity of $\forall$)

$$\begin{aligned}
= [& \forall t \in \mathcal{A}_\delta^* \bullet ((\neg(t^\frown \langle !t\rangle \in tr^\frown \delta^{*\frown} \langle ?1, !t\rangle)) \wedge \\
& \forall t \in \mathcal{A}_\delta^* \bullet ((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]
\end{aligned}$$

(definition of $\forall$)

$$= [FALSE \wedge \forall t \in \mathcal{A}_\delta^* \bullet ((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]$$

(definition of $\wedge$ and definition of $[]$)

$$= FALSE$$

As we have $\mathcal{P}_{spec} \not\sqsubseteq_{ioco} \mathcal{P}'_{iut}$ and $\mathcal{P}_{iut} \sqsubseteq \mathcal{P}'_{iut}$, because of Lemma 4.77, we can conclude $\mathcal{P}_{spec} \not\sqsubseteq_{ioco} \mathcal{P}_{iut}$.$\square$

Example 4.7. An implementation is only allowed to be quiescent after a particular trace if the specification allows this. Given the specification process $\mathcal{P}_{spec} = do_{\mathcal{A}}^\delta(?1); do_{\mathcal{A}}^\delta(!c)$ and the implementation process $\mathcal{P}_{iut} = (\iota_{\mathcal{A}}^\delta(?1); \iota_{\mathcal{A}}^\delta(!c)) +_f^\delta \iota_{\mathcal{A}}^\delta(?1)$ leads to $\mathcal{P}_{spec} \not\sqsubseteq_{ioco} \mathcal{P}_{iut}$. Again we need $\mathcal{P}'_{iut}$ which does not use the fair choice, i.e. = $\mathcal{P}'_{iut} = (\iota_{\mathcal{A}}^\delta(?1); \iota_{\mathcal{A}}^\delta(!c)) +^\delta \iota_{\mathcal{A}}^\delta(?1)$

$$\mathcal{P}_{spec} \sqsubseteq_{ioco} \mathcal{P}_{iut} =$$

(definition of $\sqsubseteq_{ioco}$)

$$\begin{aligned}
= [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet & ((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Trace(\mathcal{P}_{iut})[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_{spec})[t^\frown o/tr']) \wedge \\
& ((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Quiet(\mathcal{P}_{iut})[t/tr']) \Rightarrow Quiet(\mathcal{P}_{spec})[t/tr'])]
\end{aligned}$$

(definition of $\mathcal{P}_{spec}$, of $\mathcal{P}_i$, and of $Trace$)

$$\begin{aligned}
= [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet & (((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Trace(\mathcal{P}_{iut})[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_{spec})[t^\frown o/tr']) \wedge \\
& (((t \in tr^\frown \delta^* \ \vee\ t \in tr^\frown \delta^{*\frown} \langle ?1\rangle \ \vee\ t \in tr^\frown \delta^{*\frown} \langle ?1, !c\rangle) \wedge \\
& \ (t \in tr^\frown (\mathcal{A}_{in \setminus ?1} \cup \delta)^* \ \vee\ t^\frown o \in tr^\frown (\mathcal{A}_{in} \cup \delta)^{*\frown} \langle ?1\rangle \ \vee \\
& \ t \in tr^\frown (\mathcal{A}_{in \setminus ?1} \cup \delta)^{*\frown} \langle ?1\rangle^\frown \mathcal{A}_{in}^{*\frown} \langle !c\rangle)) \Rightarrow \\
& (t \in tr^\frown \delta^* \ \vee\ t \in tr^\frown \delta^{*\frown} \langle ?1, !c\rangle))
\end{aligned}$$

(propositional calculus)

$=[\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet (((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Trace(\mathcal{P}_{iut})[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_{spec})[t^\frown o/tr']) \wedge$
$(((\neg(t \in tr^\frown \delta^*) \wedge \neg(t \in tr^\frown \delta^{*\frown} \langle ?1 \rangle) \wedge \neg(t \in tr^\frown \delta^{*\frown} \langle ?1, !c \rangle)) \vee$
$(t \in tr^\frown \delta^* \vee t \in tr^\frown \delta^{*\frown} \langle ?1, !c \rangle)))))]$

(propositional calculus and distributivity of $\forall$)

$=[\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet ((Trace(\mathcal{P}_{spec})[t/tr'] \wedge Trace(\mathcal{P}_{iut})[t^\frown o/tr']) \Rightarrow Trace(\mathcal{P}_{spec})[t^\frown o/tr']) \wedge$
$\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet \neg(t \in tr^\frown \delta^{*\frown} \langle ?1 \rangle))]$

(definition of $\forall$ and definition of $\wedge$ and definition of [])

$= FALSE$

As we have $\mathcal{P}_{spec} \not\sqsubseteq_{ioco} \mathcal{P}'_{iut}$ and $\mathcal{P}_{iut} \sqsubseteq \mathcal{P}'_{iut}$, because of Lemma 4.77, we can conclude $\mathcal{P}_{spec} \not\sqsubseteq_{ioco} \mathcal{P}_{iut}$. □

Using the predicative definition $\sqsubseteq_{ioco}$ we can now show the relation between the input-output conformance relation and refinement.

Theorem 4.4 $\sqsubseteq \subseteq \sqsubseteq_{ioco}$

Proof 4.4. In order to prove $\sqsubseteq \subseteq \sqsubseteq_{ioco}$, we have to show that $S \sqsubseteq I \Rightarrow S \sqsubseteq_{ioco} I$.

$S \sqsubseteq I$

(definition of $\sqsubseteq$ and propositional calculus)

$= [I \Rightarrow S] \wedge ([I \Rightarrow S] \vee [I \Rightarrow S])$

(substitution)

$= [I \Rightarrow S] \wedge [(I[false/wait'] \Rightarrow S[false/wait'])] \vee [(I[A_{out}/ref'_{out}] \Rightarrow S[A_{out}/ref'_{out}])]$

(def. of [] and prop. calculus)

$= [I \Rightarrow S] \wedge [\ (I[false/wait'] \wedge I[A_{out}/ref'_{out}]) \Rightarrow (S[false/wait'] \vee S[A_{out}/ref'_{out}])]$

(propositional calculus)

$\Rightarrow [(I \Rightarrow S) \wedge ((\exists ref'_{in} \bullet (I[false/wait'] \vee I[A_{out}/ref'_{out}])) \Rightarrow$
$(\exists ref'_{in} \bullet (S[false/wait'] \vee S[A_{out}/ref'_{out}])))]$

(propositional calculus and definition of *Quiet*)

$\Rightarrow [((\exists ref, ref', wait, wait', ok, ok' \bullet I) \Rightarrow (\exists ref, ref', wait, wait', ok, ok' \bullet S)) \wedge$
$(Quiet(I) \Rightarrow Quiet(S))]$

(prop. calculus and def. of *Trace*)

$\Rightarrow [\forall t \in \mathcal{A}_\delta^* \bullet (Trace(I)[t/tr'] \Rightarrow Trace(S)[t/tr']) \wedge \forall t \in \mathcal{A}_\delta^* \bullet (Quiet(I)[t/tr'] \Rightarrow Quiet(S)[t/tr'])]$

(propositional calculus)

$\Rightarrow [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet (Trace(I)[t^\frown o/tr'] \Rightarrow Trace(S)[t^\frown o/tr']) \wedge$
$\forall t \in \mathcal{A}_\delta^* \bullet (Quiet(I)[t/tr'] \Rightarrow Quiet(S)[t/tr'])]$

(propositional calculus and definition of *Trace*)

$\Rightarrow [\ \forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet ((Trace(S)[t/tr'] \wedge Trace(I)[t^\frown o/tr']) \Rightarrow Trace(S)[t^\frown o/tr']) \wedge$
$\forall t \in \mathcal{A}_\delta^* \bullet ((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr'])]$

(distributivity of $\forall$)

$= [\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet (((Trace(S)[t/tr'] \wedge Trace(I)[t^\frown o/tr']) \Rightarrow Trace(S)[t^\frown o/tr']) \wedge$
$((Trace(S)[t/tr'] \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr']))]$

(def. of $\sqsubseteq_{ioco}$)

$= S \sqsubseteq_{ioco} I$

□

Although, the definition of $\sqsubseteq_{ioco}$ corresponds to the definition of **ioco** we can reformulate our definition to a more generic version $\sqsubseteq_{ioco}^P$ which corresponds to **ioco**$_\mathcal{F}$. Like $\mathcal{F}$ in **ioco**$_\mathcal{F}$, P is used to select the proper set of traces.

Definition 4.26 ($\sqsubseteq_{ioco}^P$) *Given an implementation process I and a specification process S, then*

$$S \sqsubseteq_{ioco}^P I =_{df} [\ \forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet$$
$$((P(S,I,t) \wedge Trace(I)[t^\frown o/tr']) \Rightarrow Trace(S)[t^\frown o/tr']) \wedge$$
$$((P(S,I,t) \wedge Quiet(I)[t/tr']) \Rightarrow Quiet(S)[t/tr'])\ ]$$

The conformance relations listed in Section 3.1 can now be defined as follows:

Definition 4.27 (Conformance relations)

$$\begin{array}{rcl}
S \sqsubseteq_{iot} I & =_{df} & S \sqsubseteq_{ioco}^{P_{iot}} I, \textit{ where } P_{iot}(S,I,t) = t \in \mathcal{A}_\delta^* \\
S \sqsubseteq_{ior} I & =_{df} & S \sqsubseteq_{ioco}^{P_{ior}} I, \textit{ where } P_{ior}(S,I,t) = t \in \mathcal{A}^* \\
S \sqsubseteq_{ioconf} I & =_{df} & S \sqsubseteq_{ioco}^{P_{ioconf}} I, \textit{ where } P_{ioconf}(S,I,t) = Trace(S)[t/tr'] \wedge t \in \mathcal{A}^* \\
S \sqsubseteq_{ioco} I & =_{df} & S \sqsubseteq_{ioco}^{P_{ioco}} I, \textit{ where } P_{ioco}(S,I,t) = Trace(S)[t/tr'] \wedge t \in \mathcal{A}_\delta^*
\end{array}$$

4.5 Test Cases, Test Processes, and Test Suites

Testing for conformance is done by applying a set of test cases to an implementation. Test cases are processes satisfying additional properties.

A test process has finite behavior such that testing can be eventually stopped. In the case of divergence one needs to interrupt testing externally.

TC1 $\quad P(tr,tr') = P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$

As a function **TC1** is written as **TC1**$(P) = P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$. This function is an idempotent.

Lemma 4.78 (TC1-idempotent)

$$\textbf{\textit{TC1}} \circ \textbf{\textit{TC1}} = \textbf{\textit{TC1}}$$

Furthermore, **TC1** commutes with **R1**, **R2**, and **IOCO1**.

Lemma 4.79 (commutativity-TC1-R1)

$$\textbf{\textit{TC1}} \circ \textbf{\textit{R1}} = \textbf{\textit{R1}} \circ \textbf{\textit{TC1}}$$

Lemma 4.80 (commutativity-TC1-R2)

$$\textbf{\textit{TC1}} \circ \textbf{\textit{R2}} = \textbf{\textit{R2}} \circ \textbf{\textit{TC1}}$$

Lemma 4.81 (commutativity-TC1-IOCO1)

$$\textbf{\textit{TC1}} \circ \textbf{\textit{IOCO1}} = \textbf{\textit{IOCO1}} \circ \textbf{\textit{TC1}}$$

A test case either accepts all responses from an implementation, i.e., inputs from the view of the test case, or it accepts no inputs at all:

TC2 $\quad P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref_{in}' = \mathcal{A}_{in} \vee ref_{in}' = \emptyset)))$

As a function **TC2** is written as **TC2**$(P) = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref_{in}' = \mathcal{A}_{in} \vee ref_{in}' = \emptyset)))$; it is an idempotent.

Lemma 4.82 (TC2-idempotent)

$$\textbf{\textit{TC2}} \circ \textbf{\textit{TC2}} = \textbf{\textit{TC2}}$$

Furthermore, **TC2** commutes with **R1**, **R2**, **IOCO1** and **TC1**.

Lemma 4.83 (commutativity-TC2-R1)

$$\mathbf{TC2} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC2}$$

Lemma 4.84 (commutativity-TC2-R2)

$$\mathbf{TC2} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC2}$$

Lemma 4.85 (commutativity-TC2-IOCO1)

$$\mathbf{TC2} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC2}$$

Lemma 4.86 (commutativity-TC2-TC1)

$$\mathbf{TC2} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC2}$$

If the test case has to provide a particular stimuli to the IUT it is always clear which output (from the view of the test case) should be sent:

TC3 $\qquad P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$

As a function **TC3** is written as **TC3**$(P) = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$. This function is an idempotent.

Lemma 4.87 (TC3-idempotent)

$$\mathbf{TC3} \circ \mathbf{TC3} = \mathbf{TC3}$$

Furthermore, **TC3** commutes with **R1**, **R2**, **IOCO1**, **TC1** and **TC2**.

Lemma 4.88 (commutativity-TC3-R1)

$$\mathbf{TC3} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC3}$$

Lemma 4.89 (commutativity-TC3-R2)

$$\mathbf{TC3} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC3}$$

Lemma 4.90 (commutativity-TC3-IOCO1)

$$\mathbf{TC3} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC3}$$

Lemma 4.91 (commutativity-TC3-TC1)

$$\mathbf{TC3} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC3}$$

Lemma 4.92 (commutativity-TC3-TC2)

$$\mathbf{TC3} \circ \mathbf{TC2} = \mathbf{TC2} \circ \mathbf{TC3}$$

Furthermore, testing should be a deterministic activity, i.e. test cases should be deterministic. Determinism includes that a tester can always deterministically decide what to do: send a particular stimuli to the IUT or wait for a possible response. This is ensured by the following two healthiness conditions.

TC4 $P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$
TC5 $P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$

These two healthiness conditions can be written as functions, i.e. $\mathbf{TC4}(P) = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ and $\mathbf{TC5}(P) = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$. Both functions are an idempotent.

Lemma 4.93 (TC4-idempotent)

$$\boldsymbol{TC4} \circ \boldsymbol{TC4} = \boldsymbol{TC4}$$

Lemma 4.94 (TC5-idempotent)

$$\boldsymbol{TC5} \circ \boldsymbol{TC5} = \boldsymbol{TC5}$$

Furthermore, both commute with **R1**, **R2**, **IOCO1**, **TC1**, **TC2** and with each other.

Lemma 4.95 (commutativity-TC4-R1)

$$\boldsymbol{TC4} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{TC4}$$

Lemma 4.96 (commutativity-TC4-R2)

$$\boldsymbol{TC4} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{TC4}$$

Lemma 4.97 (commutativity-TC4-IOCO1)

$$\boldsymbol{TC4} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{TC4}$$

Lemma 4.98 (commutativity-TC4-TC1)

$$\boldsymbol{TC4} \circ \boldsymbol{TC1} = \boldsymbol{TC1} \circ \boldsymbol{TC4}$$

Lemma 4.99 (commutativity-TC4-TC2)

$$\boldsymbol{TC4} \circ \boldsymbol{TC2} = \boldsymbol{TC2} \circ \boldsymbol{TC4}$$

Lemma 4.100 (commutativity-TC4-TC3)

$$\boldsymbol{TC4} \circ \boldsymbol{TC3} = \boldsymbol{TC3} \circ \boldsymbol{TC4}$$

Lemma 4.101 (commutativity-TC5-R1)

$$\boldsymbol{TC5} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{TC5}$$

Lemma 4.102 (commutativity-TC5-R2)

$$\boldsymbol{TC5} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{TC5}$$

Lemma 4.103 (commutativity-TC5-IOCO1)

$$\boldsymbol{TC5} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{TC5}$$

Lemma 4.104 (commutativity-TC5-TC1)

$$\boldsymbol{TC5} \circ \boldsymbol{TC1} = \boldsymbol{TC1} \circ \boldsymbol{TC5}$$

Lemma 4.105 (commutativity-TC5-TC2)

$$\boldsymbol{TC5} \circ \boldsymbol{TC2} = \boldsymbol{TC2} \circ \boldsymbol{TC5}$$

Lemma 4.106 (commutativity-TC5-TC3)

$$\boldsymbol{TC5} \circ \boldsymbol{TC3} = \boldsymbol{TC3} \circ \boldsymbol{TC5}$$

Lemma 4.107 (commutativity-TC5-TC4)

$$\boldsymbol{TC5} \circ \boldsymbol{TC4} = \boldsymbol{TC4} \circ \boldsymbol{TC5}$$

After termination a test case should give a verdict about the test execution

TC6 $\quad P = P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$

TC6 can be written as function, i.e. **TC6**$(P) = P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$; it is an idempotent.

Lemma 4.108 (TC6-idempotent)

$$\boldsymbol{TC6} \circ \boldsymbol{TC6} = \boldsymbol{TC6}$$

Furthermore, **TC6** commutes with **R1**, **R2**, **IOCO1**, **TC1**, **TC2**, **TC3**, **TC4** and **TC5**.

Lemma 4.109 (commutativity-TC6-R1)

$$\boldsymbol{TC6} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{TC6}$$

Lemma 4.110 (commutativity-TC6-R2)

$$\boldsymbol{TC6} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{TC6}$$

Lemma 4.111 (commutativity-TC6-IOCO1)

$$\boldsymbol{TC6} \circ \boldsymbol{IOCO1} = \boldsymbol{IOCO1} \circ \boldsymbol{TC6}$$

Lemma 4.112 (commutativity-TC6-TC1)

$$\boldsymbol{TC6} \circ \boldsymbol{TC1} = \boldsymbol{TC1} \circ \boldsymbol{TC6}$$

Lemma 4.113 (commutativity-TC6-TC2)

$$\boldsymbol{TC6} \circ \boldsymbol{TC2} = \boldsymbol{TC2} \circ \boldsymbol{TC6}$$

Lemma 4.114 (commutativity-TC6-TC3)

$$\boldsymbol{TC6} \circ \boldsymbol{TC3} = \boldsymbol{TC3} \circ \boldsymbol{TC6}$$

Lemma 4.115 (commutativity-TC6-TC4)

$$\boldsymbol{TC6} \circ \boldsymbol{TC4} = \boldsymbol{TC4} \circ \boldsymbol{TC6}$$

Lemma 4.116 (commutativity-TC6-TC5)

$$\boldsymbol{TC6} \circ \boldsymbol{TC5} = \boldsymbol{TC5} \circ \boldsymbol{TC6}$$

As for specifications and implementations we make $\mathbb{I}$ suitable for test cases:

$$\mathbb{I}^{TC} =_{df} \left(\begin{array}{l} \neg ok \wedge (tr \le tr') \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \le n) \wedge \\ (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)) \wedge \\ (wait' \Rightarrow (|ref'_{out}| \ge |\mathcal{A}_{out}| - 1)) \wedge \\ (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})) \wedge \\ (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})) \wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \end{array}\right) \vee \left(\begin{array}{l} ok' \wedge \\ (v' = v) \end{array}\right)$$

$\mathbb{I}^{TC}$ ensures that even in the case of divergence we respect the properties of test cases. For test cases **R3** becomes $\mathbf{R3}^{TC} =_{df} \mathbb{I}^{TC} \lhd wait \rhd P$. All test case healthiness conditions commute with $\mathbf{R3}^{TC}$.

Lemma 4.117 (commutativity-TC1-R3^{TC})

$$\boldsymbol{TC1} \circ \boldsymbol{R3}^{TC} = \boldsymbol{R3}^{TC} \circ \boldsymbol{TC1}$$

Lemma 4.118 (commutativity-TC2-R3^{TC})

$$\boldsymbol{TC2} \circ \boldsymbol{R3}^{TC} = \boldsymbol{R3}^{TC} \circ \boldsymbol{TC2}$$

Lemma 4.119 (commutativity-TC3-R3^{TC})

$$\boldsymbol{TC3} \circ \boldsymbol{R3}^{TC} = \boldsymbol{R3}^{TC} \circ \boldsymbol{TC3}$$

Lemma 4.120 (commutativity-TC4-R3^{TC})

$$\boldsymbol{TC4} \circ \boldsymbol{R3}^{TC} = \boldsymbol{R3}^{TC} \circ \boldsymbol{TC4}$$

Lemma 4.121 (commutativity-TC5-R3^{TC})

$$\boldsymbol{TC5} \circ \boldsymbol{R3}^{TC} = \boldsymbol{R3}^{TC} \circ \boldsymbol{TC5}$$

Lemma 4.122 (commutativity-TC6-R3^{TC})

$$\boldsymbol{TC6} \circ \boldsymbol{R3}^{TC} = \boldsymbol{R3}^{TC} \circ \boldsymbol{TC6}$$

Due to the properties of test cases the only choices of a test case are choices between inputs. Since the chosen input to a test case, i.e. output of the IUT, are up to the IUT this choice is given in terms of an external choice:

$$P + Q \quad =_{df} \quad P \wedge Q \lhd \mathbb{\delta} \rhd P \vee Q \;\text{ with }\; \mathbb{\delta} =_{df} \mathbf{R3}^{TC}(tr' = tr \wedge wait')$$

Definition 4.28 (Test process) *A test process P is a reactive process, which satisfies the healthiness conditions **TC1**...**TC6** and **IOCO1**. The set of events in which a test case can potentially engage is given by $\mathcal{A}$, where $\mathcal{A} = \mathcal{A}_{out} \cup \mathcal{A}_{in}$, $\mathcal{A}_{out} \cap \mathcal{A}_{in} = \emptyset$ and $\theta \in \mathcal{A}_{in}$. The observations are extended by: $pass, fail : Bool$, which denote the pass and fail verdicts, respectively.*

As in the original **ioco** theory we use θ in test processes to denote the observation of δ.

Theorem 4.5 *Test cases are closed under* $\{+,;\}$.

Proof 4.5. Idempotence of healthiness conditions and healthiness conditions are preserved (see lemmas)□

The healthiness conditions for test cases are preserved by the operators used for test cases, i.e., by sequential composition and by the external choice:

Lemma 4.123 (closure-;-TC1)

$$\boldsymbol{TC1}(P;Q) = P;Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{TC1} \textit{ healthy}$$

Lemma 4.124 (closure-;-TC2)

$$\boldsymbol{TC2}(P;Q) = P;Q \textit{ provided that } P \textit{ and } Q \textit{ are } \boldsymbol{TC2} \textit{ healthy and } Q \textit{ is } \boldsymbol{R3}_{\iota}^{\delta} \textit{ healthy}$$

Lemma 4.125 (closure-;-TC3)

$$\boldsymbol{TC3}(P;Q) = P;Q \textit{ provided } Q \textit{ is } \boldsymbol{TC3} \textit{ healthy}$$

Lemma 4.126 (closure-;-TC4)

$$\boldsymbol{TC4}(P;Q) = P;Q \textit{ provided } Q \textit{ is } \boldsymbol{TC4} \textit{ healthy}$$

Lemma 4.127 (closure-;-TC5)

$$\boldsymbol{TC5}(P;Q) = P;Q \textit{ provided } Q \textit{ is } \boldsymbol{TC5} \textit{ healthy}$$

Lemma 4.128 (closure-;-TC6)

$$\boldsymbol{TC6}(P;Q) = P;Q \textit{ provided } Q \textit{ is } \boldsymbol{TC6} \textit{ healthy}$$

Lemma 4.129 (closure-+-IOCO1)

$$\boldsymbol{IOCO1}(P+Q) = P+Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{IOCO1} \textit{ healthy}$$

Lemma 4.130 (closure-+-TC1)

$$\boldsymbol{TC1}(P+Q) = P+Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{TC1} \textit{ healthy}$$

Lemma 4.131 (closure-+-TC2)

$$\boldsymbol{TC2}(P+Q) = P+Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{TC2} \textit{ healthy}$$

Lemma 4.132 (closure-+-TC3)

$$\boldsymbol{TC3}(P+Q) = P+Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{TC3} \textit{ healthy}$$

Lemma 4.133 (closure-+-TC4)

$$\boldsymbol{TC4}(P+Q) = P+Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{TC4} \textit{ healthy}$$

Lemma 4.134 (closure-+-TC5)

$$\boldsymbol{TC5}(P+Q) = P+Q \textit{ provided } P \textit{ and } Q \textit{ are } \boldsymbol{TC5} \textit{ healthy}$$

Lemma 4.135 (closure-$+$-TC6)

$$\textbf{\textit{TC6}}(P+Q) = P+Q \text{ provided } P \text{ and } Q \text{ are } \textbf{\textit{TC6}} \text{ healthy}$$

Test cases are reactive processes expressed in terms of $do_{\mathcal{A}}(a)$. We use the following abbreviations for indicating pass (✓) and fail (✗) verdicts.

$$✓ =_{df} (\neg wait' \Rightarrow pass') \qquad ✗ =_{df} (\neg wait' \Rightarrow fail')$$

Example 4.8. For example, a valid test process $\mathcal{P}_T$ with the alphabet $A_{in} = \{c, t, \theta\}$ and $A_{out} = \{1\}$ that sends a stimulus and subsequently accepts a c-response but neither accepts t nor θ (see test case T of Figure 3.4) is given by

$$\mathcal{P}_T = !1; ((?c \wedge ✓) + (?t \wedge ✗) + (\theta \wedge ✗)) = (\text{def. of } do_{\mathcal{A}}, ✓, ✗, \text{ and } +)$$
$$= \begin{pmatrix} !1 \notin ref' \wedge tr' = tr \vee \\ \{?c, ?t, \theta\} \not\subseteq ref' \wedge tr' = tr ^\frown \langle !1 \rangle \end{pmatrix} \lhd wait' \rhd \begin{pmatrix} tr' = tr ^\frown \langle !1, ?c \rangle \wedge pass' \vee \\ tr' = tr ^\frown \langle !1, ?t \rangle \wedge fail' \vee \\ tr' = tr ^\frown \langle !1, \theta \rangle \wedge fail' \end{pmatrix}$$

□

A test suite is a set of test cases. Because of the use of global verdicts (see Definition 4.32), a test suite is given by the nondeterministic choice of a set of test cases.

Definition 4.29 (Test suite) *Given a set of N test processes $T_1, \ldots, T_N$, then a test suite TS is defined as: $TS =_{df} \bigsqcap_{i=1,\ldots,N} T_i$*

4.6 Testing Implementations

Test case execution in the input-output conformance testing framework is modeled by executing the test case parallel to the system under test. We model this parallel execution again as parallel by merge (see Section 4.2).

The execution of a test case t on an implementation i is denoted by $t \rceil|_U\ i$. This new process $t \rceil|_U\ i$ consists of all traces present in both, the test case and the implementation. Furthermore, $t \rceil|_U\ i$ gives $fail'$ and $pass'$ verdicts after termination.

Such an execution operator is inspired by CSPs parallel composition (Hoare, 1985), i.e, the parallel composition of a test case and an implementation can only engage in a particular action if both processes participate in the communication.

Since a test case swaps inputs and outputs of the IUT we need to rename alphabets. Therefore, we define an alphabet renaming operator for a process P denoted by $\overline{P}$ as follows: $\mathcal{A}\overline{P} =_{df} \mathcal{A}P$; $\mathcal{A}_{out}\overline{P} =_{df} \mathcal{A}_{in}P$; $\mathcal{A}_{in}\overline{P} =_{df} \mathcal{A}_{out}P$. In addition, $\overline{\theta} =_{df} \delta$.

Definition 4.30 (Test case execution) *Let TC be a test case process and IUT be an implementation process, then*

$$\mathcal{A}(TC \rceil|_U\ IUT) =_{df} \mathcal{A}\overline{TC} \cup \mathcal{A}IUT$$
$$TC \rceil|_U\ IUT =_{df} (\overline{TC} \curlywedge IUT); M_{ti}$$

The relation M_{ti} merges the traces of the test case and the implementation. The result comprises the pass and fail verdicts of the test case as well as traces that are allowed in both, the test case and the implementation. Because of our representation of quiescence, there is no δ that indicates termination of the IUT, i.e., $\neg 1.wait$. M_{ti} takes care of that when merging the traces.

Definition 4.31 (Test case/impl. merge)

$$\begin{aligned} M_{ti} \quad =_{df} \quad & pass' = 0.pass \wedge fail' = 0.fail \wedge wait' = (0.wait \wedge 1.wait) \wedge \\ & ref' = (0.ref \cup 1.ref) \wedge ok' = (0.ok \wedge 1.ok) \wedge \\ & (\exists u \bullet ((u = (0.tr - tr) \wedge u = (1.tr - tr) \wedge tr' = tr^\frown u) \vee \\ & \quad (u^\frown \mathcal{A}_{in}^{*\frown} \langle \delta \rangle = (0.tr - tr) \wedge u = (1.tr - tr) \wedge tr' = tr^\frown \langle u, \delta \rangle) \wedge \neg 1.wait)) \end{aligned}$$

Due to the lack of symmetry of our merge operator the test case execution operator $\rceil|_U$ is not symmetric. However, it still satisfies one important property. Let T_1, T_2 be test cases and let P be an implementation, then

Lemma 4.136 ($\rceil|_U$ -$\sqcap$-distributivity)

$$(T_1 \sqcap T_2)\rceil|_U\ P = (T_1\rceil|_U\ P) \sqcap (T_2\rceil|_U\ P)$$

This law allows one to run a set of N test cases $T_1, \ldots, T_N$, i.e. a test suite $TS =_{df} \bigsqcap_{i=1,\ldots,N} T_i$, against an implementation process P:

$$TS\rceil|_U\ P = \bigsqcap_{i=1,\ldots,N} T_i\rceil|_U\ P = (T_1 \sqcap \ldots \sqcap T_N)\rceil|_U\ P$$

Since our test cases do not consist of a single trace but of several traces there may be different verdicts given at the end of different traces. An implementation passes a test case if all possible test runs lead to the verdict pass:

Definition 4.32 (Global verdict) *Given a test process (or a test suite) T and an implementation process IUT, then*

$$\begin{aligned} IUT\,passes\,T \quad &=_{df} \quad \forall r \in \mathcal{A}_\delta^* \bullet (((T\rceil|_U\ IUT)[r/tr'] \wedge \neg wait') \Rightarrow pass') \\ IUT\,fails\,T \quad &=_{df} \quad \exists r \in \mathcal{A}_\delta^* \bullet (((T\rceil|_U\ IUT)[r/tr'] \wedge \neg wait') \Rightarrow fail') \end{aligned}$$

Example 4.9. Now we can calculate verdicts by executing test cases on implementations. For example, consider the test case of Example 8, i.e. $\mathcal{P}_T =!1;((?c \wedge ✓) + (?t \wedge ✗) + (\theta \wedge ✗))$, and the IUT $\mathcal{P}_i = \iota_{\mathcal{A}}^{\delta}(?1);\iota_{\mathcal{A}}^{\delta}(!c)$ (representing the IUT i of Figure 3.3). Executing $\mathcal{P}_T$ on the IUT $\mathcal{P}_i$, i.e. $\mathcal{P}_T\rceil|_U\ \mathcal{P}_i$, is conducted as follows:

$$\begin{aligned} & \mathcal{P}_T\rceil|_U\ \mathcal{P}_i = && \text{(def. of } \rceil|_U \text{ and renaming)} \\ & = (?1;((!c \wedge ✓) + (!t \wedge ✗) + (\theta \wedge ✗)) \curlywedge \iota_{\mathcal{A}}^{\delta}(?1);\iota_{\mathcal{A}}^{\delta}(!c));M_{ti} && \text{(def. of } \curlywedge \text{ and } M_{ti}) \\ & = \begin{pmatrix} ?1 \notin ref' \wedge tr' = tr \vee \\ !c \notin ref' \wedge tr' = tr^\frown \langle ?1 \rangle \end{pmatrix} \lhd wait' \rhd (tr' = tr^\frown \langle ?1, ?c \rangle \wedge pass') \end{aligned}$$

Thus, we have $\mathcal{P}_i$ passes $\mathcal{P}_T$ because

$$\begin{aligned} & \mathcal{P}_i \text{ passes } \mathcal{P}_T && \text{(def. of passes)} \\ & = \forall r \in \mathcal{A}_\delta^* \bullet (((\mathcal{P}_T\rceil|_U\ \mathcal{P}_i)[r/tr'] \wedge \neg wait') \Rightarrow pass') && (t\rceil|_U\ \mathcal{P}_i) \\ & = \forall r \in \mathcal{A}_\delta^* \bullet ((\neg wait' \wedge r = tr^\frown \langle ?1, !c \rangle \wedge pass') \Rightarrow pass') && \text{(prop. calc.)} \\ & = \forall r \in \mathcal{A}_\delta^* \bullet TRUE \\ & = TRUE \end{aligned}$$

□

Chapter 5

Asynchronous Input-Output Conformance Testing

The contents of this chapter have been published in the paper (Weiglhofer and Wotawa, 2009a).

As the theory of the previous chapter shows, the models used for input-output conformance testing have to obey certain restrictions. Specifications have to satisfy the healthiness conditions **IOCO1** (exclusion of live-locks), **IOCO2**, and **IOCO3** (both describe the occurrence of quiescence). The implementations that can be tested with respect to input-output conformance have to be **IOCO1**-**IOCO5** healthy. The test cases that are used for assessing the (ioco-) correctness of an implementation with respect to its specification have to satisfy the healthiness conditions **TC1**-**TC6**, and **IOCO1**.

The healthiness condition **TC2** (a test case may accept no inputs) together with the test execution operator, i.e. $\rceil|_U$, have an awkward effect. When applying test cases to real implementations, the test cases sometimes need to block outputs of the implementation. In other words, test cases may prevent implementations from providing outputs. While this has already been noticed by others (e.g. Tan and Petrenko (1998); Petrenko and Yevtushenko (2002)), it is also highlighted by the denotational version of **ioco**, especially by the healthiness conditions.

The healthiness condition **TC2** ensures that a test case either accepts all inputs (outputs of an implementation) or no inputs at all, i.e.,

$$P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$$

As one can see from **TC2** a test case may refuse to do any input, i.e. it may refuse to accept any output from an implementation: $ref'_{in} = \mathcal{A}_{in}$.

Recall the merge relation that is used for formalizing the test case execution operator (Definition 4.31):

$$M_{ti} = \cdots \wedge ref' = (0.ref \cup 1.ref) \wedge \ldots$$

This merge relation is used to combine the traces of an implementation and a test case during test case execution. Due to the union of the refusal sets, the resulting refusal set of the process representing the test case execution comprises all inputs as well. Thus, an implementation is not allowed to provide an output, if a test case refuses to synchronize on inputs (outputs of the implementation), i.e. $ref'_{in} = \mathcal{A}_{in}$.

While this is not a problem if test case execution is conducted synchronously blocking outputs of an implementation may be challenging in asynchronous environments. In an asynchronous environment messages between a tester and the implementation are queued. This may lead to incorrect verdicts. In practice

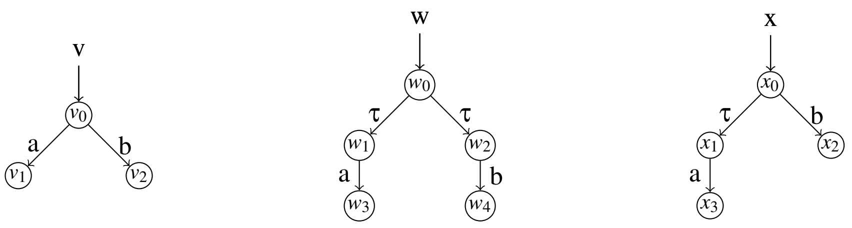


Figure 5.1: Different types of choices.

asynchronous communication occurs when for example the communication between a tester and the implementation is implemented by the use of TCP or UDP. A test case and an implementation that communicate via TCP do not directly synchronize on their actions. Instead test cases and implementations communicate asynchronously via queues.

In this chapter we discuss how restricting the used choices helps to relax some of the input-output conformance testing assumptions. We require that only the IUT is allowed to choose whether to send an output or to wait for an input; the environment cannot influence this choice. Thus, choices between inputs and outputs are internal choices of the IUT. In that case, we can use the observation of quiescence, i.e. observing the absence of outputs, as a handshake between a test case and an IUT. By this restriction we gain the following benefits:

- Test cases can test for **ioco** in asynchronous environments;
- The assumption about input enabledness of implementations is relaxed; and
- A test case never has to choose between sending a stimulus or waiting for a response. It is always clear what a tester has to do.

5.1 Internal and External Choices

An internal choice is a choice where the system itself decides which actions are offered to the environment. On the contrary, external choices allow the environment to choose between different actions.

Example 5.1. The LTSs in Figure 5.1 illustrate the difference between internal and external choices. The LTS v shows an external choice between the actions a and b. That is, in the state v_0 the environment can choose whether to synchronize on a or on b. On the contrary, w depicts an internal choice between a and b. That is, the system may internally decide to move to state w_1 or to state w_2. In any case only one action is offered for communication. Also the LTS x depicts an internal choice. The system may choose to offer an a without b being an option for the environment. □

If there is a choice between inputs and outputs the type of choice affects quiescence. In the case of an external choice between an input $a \in L_I$ and an output $b \in L_U$ we cannot observe quiescence. On the contrary, if we have an internal choice between $a \in L_I$ and $b \in L_U$ we can observe quiescence before a. However, there is one exception. We can observe quiescence only if there are no τ transitions enabled. Thus, if an implementation can choose between waiting for input a or internally moving to a state where output b is offered we cannot observe quiescence.

Example 5.2. The observability of quiescence when having different types of choices is illustrated in Figure 5.2. We can observe quiescence in state x'_1 but not in state x''_1. This is due to the fact that, in the LTS x'', the system may decide to move to x''_1 without $?a$ being an option for the environment. □

By restricting the choices of LTSs to choices where quiescence can be observed, quiescence can be seen as an observation that an implementation is willing to accept inputs. We call such a labeled transition system internal choice input-output labeled transition system.

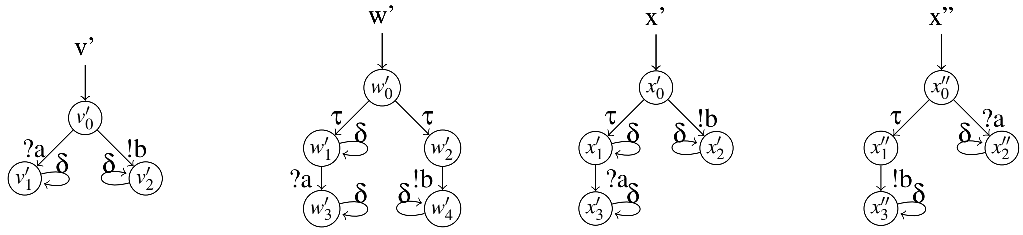


Figure 5.2: Choices affect quiescence.

Definition 5.1 (internal choice LTS) *An LTS $M = (Q, L_I \cup L_U \cup \{\tau, \delta\}, \rightarrow, q_0)$ is an internal choice LTS iff $\forall q \in Q, \exists a \in L_I \bullet a \in init(q)$ implies $\delta(q)$*

The class of internal choice LTS with internal choices ($\sqcap$), such that quiescence (δ) can be observed before inputs, is given by $\mathcal{LTS}^{\sqcap}_{\delta}(L_I, L_U)$, with $\mathcal{LTS}^{\sqcap}_{\delta}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

In terms of our UTP theory, this means that we have an additional healthiness condition for specifications and implementations:

IOCO6 $\qquad P = P \wedge (\neg wait \Rightarrow wait' \Rightarrow (ref'_{in} \neq A_{in}) \Rightarrow (\theta \notin ref'))$

5.2 Relaxing Input-enabledness

Having an observation that denotes the will to accept inputs allows us to relax the assumption of input-enabledness. As our test cases ensure that they provide a stimulus only if the implementation is willing to accept a stimulus we do not need to assume that implementations are input-enabled. An implementation only needs to accept all possible inputs in states where δ can be observed.

Thus, our models (internal choice IOTS) for implementations differ from input-output transition systems.

Definition 5.2 (internal choice IOTS) *An internal choice input-output transition system is an internal choice LTS $M = (Q, L_I \cup L_U \cup \{\tau, \delta\}, \rightarrow, q_0)$ where all input actions are enabled (possibly preceded by τ transitions) in quiescent states: $\forall a \in L_I, \forall q \in Q \bullet (\delta(q)$ implies $q \overset{a}{\Longrightarrow})$*

The class of internal choice IOTS with inputs L_I and outputs L_U is given by $\mathcal{IOTS}^{\sqcap}_{\delta}(L_I, L_U)$. Note that $\mathcal{IOTS}^{\sqcap}_{\delta}(L_I, L_U) \nsubseteq \mathcal{IOTS}(L_I, L_U)$ because internal choice IOTS do not require to be input-enabled in states having outgoing transitions labeled with output actions.

Input-enabledness is expressed by our healthiness condition **IOCO5**. For internal choice IOTSs this healthiness can be weakened. We will call the weakened healthiness condition **IOCO5**$_w$:

IOCO5$_w$ $\qquad P = P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\theta \notin ref') \Rightarrow (ref'_{in} = \emptyset)))$

Example 5.3. Figure 5.3 shows the internal choice IOTSs for the LTSs of Figure 3.2. As there can be seen, these IOTSs are only input-enabled in quiescent states, e.g., a is input-enabled in the states a_0, a_2, a_3 and a_5 but not in a_1 and a_4. $\square$

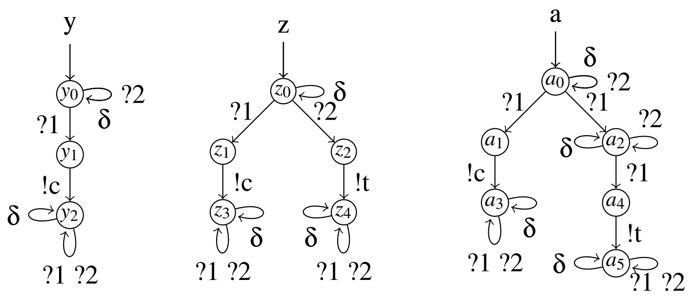


Figure 5.3: Examples of internal choice IOTS.

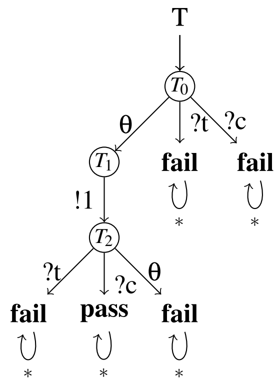


Figure 5.4: Example of a test case.

5.3 Test Cases and Test Execution

Test cases use the observation of quiescence as information that the IUT waits for inputs. Consequently, test cases only provide a stimulus if the IUT is willing to accept a stimulus, which is after observing a δ.

Definition 5.3 (internal choice test case) *An internal choice test case T is an LTS $T = (Q, L_U \cup L_I \cup \{\theta\}, \rightarrow, q_0)$ such that (1) T is deterministic and has finite behavior; (2) Q contains terminal states **pass** and **fail**; (3) for any state $q \in Q$ where $q \neq$ **pass** and $q \neq$ **fail**, either $init(q) = \{a\}$ for some $a \in L_U$, or $init(q) = L_I \cup \{\theta\}$; (4) states with $init(q) = \{a\}$ with some $a \in L_U$ are only reachable by θ-labeled transitions.*

$\mathcal{TTS}_\delta^\sqcap(L_U, L_I)$ denote the class of internal choice test cases with inputs L_I and outputs L_U.

In addition to **TC1**-**TC6**, test cases for testing internal choice IOTS obey one more healthiness condition. This healthiness condition, which we call **TC7**, formalizes that states with an output are only reachable by θ-labeled transitions:

$$\textbf{TC7} \qquad P = P \wedge ((tr' - tr) \in \mathcal{A}_\theta^{*} \frown \theta \frown \mathcal{A}_{out} \vee (tr' - tr) \in \mathcal{A}_\theta^{*} \frown \mathcal{A}_{in})$$

Example 5.4. Figure 5.4 shows an internal choice test case. In a state either inputs (outputs of the IUT) and the θ event are enabled, or an output is enabled. Outputs are only enabled in states reachable by θ-labeled transitions. Verdicts are given in terminal states of the test case. □

Testing is conducted by running a test case $t \in \mathcal{TTS}_\delta^\sqcap(L_U, L_I)$ against an IUT $i \in \mathcal{IOTS}_\delta^\sqcap(L_I, L_U)$. In a test run inputs to the IUT are under control of the tester, while outputs are under control of the IUT.

As usual we employ a fairness assumption on our implementations, i.e., an IUT will eventually show all its possible behaviors when it is re-executed with the same set of inputs. Thus if there is an erroneous output from an implementation we will eventually come up with a fail verdict.

Internal choice test cases allow to test implementations synchronously (see Section 3.1.1), and to test implementations in asynchronous environments.

5.4 Asynchronous Test Execution

In an asynchronous environment the events of the test case t and the implementation i do not synchronize directly but are passed between t and i through FIFO queues. We do not allow the environment to change the order of the sent messages. However, inputs and outputs may interleave, thus we have one input buffer σ_i and one output buffer σ_u.

A message sent by the IUT is not directly delivered to the tester but is put into the queue. Putting an element into the queue is unobservable (I3 of Def. 5.4). Eventually, the tester will pick the elements from the queue. Thus the message becomes observable (A2). A tester can send inputs at any moment in time, i.e. it puts the stimulus into the input queue (A1). Eventually, the IUT receives the event (I2). Unobservable transitions stay unobservable (I1).

Formally, we use a queue operator Verhaard et al. (1993) in order to obtain the observable behavior of an IUT in the context of queues.

Definition 5.4 (queue operator) *Let $\sigma_i \in L_I{}^*$ and $\sigma_u \in L_U{}^*$, then the unary queue operator $[{}_{\ll\sigma_u} \cdot {}_{\gg\sigma_i}] : \mathcal{LTS}(L_I, L_U) \rightarrow \mathcal{LTS}(L_I, L_U)$ is defined by the following two axioms*

$$\left[{}_{\sigma_u\ll}q_{\ll\sigma_i}\right] \xrightarrow{a} \left[{}_{\sigma_u\ll}q_{\ll\sigma_i\frown a}\right],\ a \in L_I \qquad \text{(A1)}$$

$$\left[{}_{x\frown\sigma_u\ll}q_{\ll\sigma_i}\right] \xrightarrow{x} \left[{}_{\sigma_u\ll}q_{\ll\sigma_i}\right],\ x \in L_U \qquad \text{(A2)}$$

and by the following inference rules:

$$\frac{q \xrightarrow{\tau} q'}{\left[{}_{\sigma_u\ll}q_{\ll\sigma_i}\right] \xrightarrow{\tau} \left[{}_{\sigma_u\ll}q'_{\ll\sigma_i}\right]} \qquad \text{(I1)}$$

$$\frac{q \xrightarrow{a} q',\ a \in L_I}{\left[{}_{\sigma_u\ll}q_{\ll a\frown\sigma_i}\right] \xrightarrow{\tau} \left[{}_{\sigma_u\ll}q'_{\ll\sigma_i}\right]} \qquad \text{(I2)}$$

$$\frac{q \xrightarrow{x} q',\ x \in L_U}{\left[{}_{\sigma_u\ll}q_{\ll\sigma_i}\right] \xrightarrow{\tau} \left[{}_{\sigma_u\frown x\ll}q'_{\ll\sigma_i}\right]} \qquad \text{(I3)}$$

The initial state of a queue context containing an LTS S is given by $\mathcal{Q}(S) =_{df} \left[{}_{\langle\rangle\ll}q_0{}_{\ll\langle\rangle}\right]$.

The queue operator does not consider quiescence δ. This is because quiescence denotes the absence of outputs, which is observed by not receiving outputs for a certain time span. This is not an event that an implementation can put into the queue. In a queued context quiescence is observed when there are no successor states (reachable by τ-labeled transitions) with enabled output actions.

Definition 5.5 *Let $\mathcal{Q}(M) = (Q, L_I \cup L_U \cup \{\tau\}, \rightarrow, q_0)$ be an LTS within a queue context then a state $q \in Q$ is quiescent, denoted by $\delta(q)$, iff $\forall a \in L_U \bullet a \notin init(q$ **after** $\varepsilon)$.*

The τ transition between putting an output into the queue and the reception of the output can be interpreted as message delivery. Thus, from the definition of quiescence one can see that the timeout for detecting quiescence needs to be large enough such that the message delivery is ensured within that time.

Testing an implementation i with a test case t in an asynchronous environment is then denoted by $t\rceil|Q(i)$. While $Q(i)$ can perform traces that i cannot perform, the δ event allows us to ensure that test runs only comprise traces of i. Because we do not provide inputs before observing quiescence there is no interleaving between inputs and outputs.

Example 5.5. Consider the test case T of Figure 5.4 and the IUT y of Figure 5.3 in an asynchronous context, i.e., $Q(r)$. The only possible test run is

$$T_0\rceil|\left[_{\langle\rangle\ll}y_{0\ll\langle\rangle}\right] \overset{\langle\theta,!1,?c\rangle}{\Longrightarrow} \mathbf{pass}\rceil|\left[_{\langle\rangle\ll}y_{2\ll\langle\rangle}\right]$$

which leads to a pass verdicts, and thus we have $Q(r)$ **passes** T. $\square$

As stated by the following theorem, the results of testing an implementation $i \in \mathcal{IOTS}_\delta^{\sqcap}(L_I, L_U)$ with a test case $t \in \mathcal{TTS}_\delta^{\sqcap}(L_U, L_I)$ are the same in both environments, the synchronous and the asynchronous environment:

Theorem 5.1 *Let $i \in \mathcal{IOTS}_\delta^{\sqcap}(L_I, L_U)$ and $t \in \mathcal{TTS}_\delta^{\sqcap}(L_U, L_I)$, then i **passes** $t \iff Q(i)$ **passes** t*

Sketch of proof. Due to our restriction of choices, quiescence can be observed before every input. By definition of quiescence, an implementation will not send any output before receiving an input after quiescence has been observed. The structure of our test cases guarantee that inputs to the IUT are only provided after observing δ. Thus there is no interleaving between inputs and outputs. The additional behavior of an implementation i in a queued context, i.e., $Q(i)$, is caused by interleaving. A test case t prevents interleaving of inputs and outputs, thus t only selects the traces allowed by i from $Q(i)$. Therefore, $t\rceil|Q(i)$ and $t\rceil|i$ lead to the same set of traces. By the definition of **passes** this leads to i **passes** $t \iff Q(i)$ **passes** t.

5.5 Test Case Generation

Given a specification S we like to generate test cases that possibly find non-conforming implementations. According to the definition of **ioco** we have to check whether the outputs of the implementation are allowed by the specification after a particular trace of S.

The idea of the algorithm is to generate a test case t that behaves like a particular $\sigma \in Straces(S)$. Executing t on an implementation i leads to $t\rceil|i \overset{\sigma}{\Longrightarrow} t'\rceil|i'$. At the state t' the test case needs to check if the outputs of the implementation are allowed by S. This is done by having transitions from t' to **pass** verdict states for allowed outputs o, i.e., $o \in out(s \textbf{ after } \sigma)$, and transitions from t' to **fail** verdict states for erroneous outputs o', i.e., $o' \notin out(s \textbf{ after } \sigma)$.

Additionally, the test case generation algorithm takes care that inputs to the implementation are provided only if the implementation is willing to accept inputs. That is, a test case provides a stimuli only after observing δ. A test case does not accept any response from the implementation when providing a stimuli, because an IUT will not provide outputs after a δ is observed.

Algorithm 5.1 (GENERATETC) allows one to obtain test cases, which correspond to Definition 5.3. For describing the algorithm we use some of the behavior expressions of Tretmans (2008). Basically, we need three operators ($;$, Σ, $\Box$) to form behavior expressions. Let $a \in L_I \cup L_U \cup \{\theta\}$, B_1, B_2 be some behavior expression and $\mathcal{B}$ be a countable set of behavior expressions, then $a;B$ defines the behavior which can first perform a and then behaves as B. $\Sigma\mathcal{B}$ denotes a choice of behavior, i.e., it behaves as one of the behaviors of $\mathcal{B}$. We use $B_1 \Box B_2$ as an abbreviation for $\Sigma\{B_1, B_2\}$.

Algorithm 5.1 Test Case Generation

```
1: procedure GENERATETC($s$)
2:     return pass
3:     $\sqcap$
4:     return
           $\Sigma\{x_j;\mathbf{fail} | x_j \in L_U \wedge x_j \notin out(s)\}$
           $\Box$  $\Sigma\{\theta;\mathbf{fail} | \delta \notin out(s)\}$
           $\Box$  $\Sigma\{x_i;\text{GENERATETC}(s \textbf{ after } x_i) |$
                        $x_i \in L_U \wedge x_i \in out(s)\}$
           $\Box$  $\Sigma\{\theta;\text{ADDSTIMULUS}(s \textbf{ after } \delta) | \delta \in out(s)\}$
5: end procedure

6: procedure ADDSTIMULUS($s$)
7:     if $\exists a \in L_I \bullet (s \textbf{ after } a \neq \emptyset)$ then
8:         $a \leftarrow \sqcap\{a | a \in L_I \wedge (s \textbf{ after } a \neq \emptyset)\}$
9:         $t_a \leftarrow \text{GENERATETC}(s \textbf{ after } a)$
10:        return $a;t_a$
11:    else
12:        return pass
13:    end if
14: end procedure
```

GENERATETC takes a set of states and returns a tree structured test case. Initially, for a specification $S \in \mathcal{LTS}_\delta^{\sqcap}(L_I, L_U)$ the set of states is given by S **after** ε.

Then, the two steps (Line 2 and Line 4) of Algorithm 5.1 are chosen non-deterministically ($\sqcap$). Either the test case is terminated by a **pass** verdict, or the test case offers to synchronize on all outputs $x_j \in L_U$. Outputs x_j that are not allowed by the specification, i.e. $x_j \notin out(s)$ lead to **fail** verdicts. If quiescence is not allowed by the specification, the θ event (observation of δ) leads to **fail** as well.

We add one edge for every output x_i allowed by the specification ($x_i \in out(s)$). The subtree after x_i is generated by calling the test case generation algorithm recursively with the set of states s **after** x_i. If δ is observed, then the implementation is willing to accept an input. The subtree attached to an θ-labeled edge is generated by ADDSTIMULUS.

The procedure ADDSTIMULUS checks if there is a stimulus a (possible preceded by τ-labeled transitions) which is allowed by the specification, i.e., s **after** $a \neq \emptyset$. If there exists one or more such stimuli, then this procedure selects one stimulus non-deterministically (Line 8). The sub-tree after that stimulus is generated by GENERATETC. If there is no stimulus then testing is terminated with a **pass** verdict.

A test case generation algorithm has to be sound and (at least theoretically) exhaustive. That is, the generated test cases should not give **pass** verdicts on non-conforming IUTs (soundness). A test case generation algorithm is exhaustive if it is able to detect any non-conforming IUT.

As with the original **ioco** relation, we assume that implementations are fair , i.e., an IUT will eventually show its possible behaviors when it is re-executed with the same set of inputs. Due to that assumption our algorithm allows for (theoretically) exhaustive testing. Furthermore, we do not generate test cases that fail on **ioco** correct implementations, i.e. the algorithm is sound.

5.6 A Practical Example

In this section we motivate the need of asynchronous testing by showing the results obtained when applying our approach to a Conference Protocol* (see also Section 8.1).

*http://fmt.cs.utwente.nl/ConfCase/

Table 5.1: Test execution results.

IUT	Test Runs	TORX			ALGORITHM 5.1	
		Pass	Fail	Viol.	Pass	Fail
Correct	100	18	0	82	100	0
Mutants	2700	550	155	1995	2443	257

We compared testing with our test case generation algorithm to testing with the TORX tool (Tretmans and Brinksma, 2003). TORX uses the algorithm presented by Tretmans (1996) in order to obtain test cases as defined by Definition 3.10. For our evaluation we used one correct and 27 erroneous implementations. The specification and also the implementations satisfied our assumption on choices between inputs and outputs.

The conference protocol relies on UDP for communication. In order to ensure that messages are delivered in order and that there are no messages lost we ran the test cases on the same host as the IUT. As the used implementations react quickly we used a timeout of 500ms for the observation of quiescence.

Table 5.1 lists the results obtained by TORX and by our approach on the different implementations of the protocol. In both cases, we relied on randomness for test case selection and we ran each algorithm 100 times on every IUT. We limited the test sequence length to at most 25 actions.

Table 5.1 lists the total number of test runs (2nd column) for the correct (first row) and for the erroneous (second row) implementations. This table shows the total number of obtained pass verdicts (3rd and 6th column) and the total number of fail verdicts (4th and 7th column) for which no assumptions have been violated. In addition, the 5th column lists the number of test runs for which TORX failed to establish the necessary assumptions, i.e., outputs of the system should have been blocked.

As Table 5.1 shows, TORX failed to establish the assumptions of the **ioco** relation in approx. 74% of our test runs. In other words, in 2077 out of the 2800 test runs TORX opted for sending a stimulus to the IUT while the IUT chose to provide an output. The verdicts of these test runs are unreliable. Ignoring the output of the IUT led to 75 fail verdicts and 7 additional pass verdicts on the correct IUT.

On the contrary, internal choice test cases always resulted in valid test runs. By the use of our approach combined with random test case selection we detected 19 of the 27 erroneous implementations. In contrast to that, the reliable test runs of the TORX tool only detected 12 erroneous implementations.

Note, that for testing with TORX Belinfante et al. (1999) extended the Conference Protocol specification with a model of the asynchronous environment, i.e. first-in/first-out (FIFO) buffers represent the model's context. Because of these buffers TORX was applicable for testing the Conference Protocol even in an asynchronous environment. However, the state space of the Conference Protocol specification with these buffer was much larger than the state space of the specification without these buffers.

Chapter 6

Test Purpose Based Test Case Selection

Parts of the contents of this chapter have been published in (Aichernig et al., 2007; Fraser et al., 2008b; Weiglhofer and Wotawa, 2008a; Weiglhofer et al., 2009a; Fraser et al., 2008a; Weiglhofer et al., 2009)

While the previous chapters have presented a theoretical look on input-output conformance testing, a major issue for applying the theory in practice is the selection of a proper set of test cases. One promising technique is the use of test purposes for test case selection. Test purposes can be used to create test cases efficiently, even when large models are used for test case generation. A test purpose is a formal representation of what the tester intends to exercise. Given a test purpose, only a limited part of a test model needs to be considered which attenuates the state space explosion problem.

Currently, test purposes are mostly hand written. This still leaves the tester with the task of writing test purposes, which might turn out rather difficult if thorough testing is required. In fact, despite all formal conformance relations, the real quality of the test suites depends on the ability of the tester to formulate suitable test purposes. For example, du Bousquet et al. (2000) report that even after ten hours of manual test purpose design they failed to find a set of test purposes that would detect all mutants of a given implementation.

In this chapter we investigate three techniques that assist a test engineer by automating the generation of test purposes and by complementing manual test purposes:

- We present a technique for extracting multiple test cases for a single test purpose. Currently, often one test case per test case is used.
- Second, we show how to apply mutation based test purposes to industrial systems. Test cases derived from mutation based test purposes target particular faults of the high level specification.
- The third approach deals with coverage based test purposes. The idea is to have a set of test purposes that leads to test cases which cover important aspects of the high level specification.

6.1 Test Purpose based Test Case Generation

A test purpose can be seen as a formal specification of a test case. Tools like SAMSTAG (Grabowski et al., 1993), TGV (Jard and Jéron, 2005) and Microsoft's SPECEXPLORER (Veanes et al., 2008) use test purposes for test generation. As our approaches rely on TGV we use TGV's notion of test purposes. Thus, test purposes are defined as LTSs:

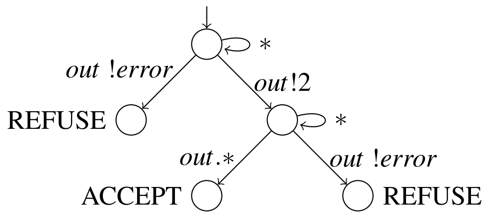


Figure 6.1: Example test purpose.

Definition 6.1 (Test Purpose) *Given a specification S in the form of an LTS, a test purpose is a deterministic LTS $TP = (Q, L, \rightarrow, q_0)$, equipped with two sets of trap states: $Accept^{TP}$ defines pass verdicts, and $Refuse^{TP}$ limits the exploration of the graph S. Furthermore, TP is complete (i.e., it allows all actions in each state).*

Example 6.1. Figure 6.1 serves to illustrate a test purpose and the used notation for test purposes in this thesis. We use '$*$'-labeled edges to denote all edges that are not explicitly specified for a particular state. Edge labels that end with '$.*$' denote any edge that starts with the text stated before '$.*$'. The illustrated test purpose is intended for our stack calculator specification of Figure 3.11. This test purpose selects traces of the specification's LTS that end with ⟨`out!2, out.*`⟩. It refuses to select any trace that contains `out!error`. □

According to Jard and Jéron (2005) test synthesis within TGV is conducted as follows: Given a test purpose *TP* and a specification *S*, TGV calculates the synchronous product $SP = S \times TP$. The construction of *SP* is stopped in *Accept* and *Refuse* states as subsequent behaviors are not relevant to the test purpose. Then TGV marks all states where neither an output nor a τ-labeled transition is possible by adding δ labeled self-loops to these states (c.f. Definition 3.4). Before a test case is extracted, TGV obtains the observable behavior SP^{VIS} by making *SP* deterministic. Note that SP^{VIS} does not contain any τ-labeled transitions.

A test case derived by TGV is controllable, i.e., it does not have to choose between sending different stimuli and between waiting for responses and sending stimuli. This is achieved by selecting traces from SP^{VIS} that lead to *Accept* states and pruning edges that violate the controllability property. Finally, the states of the test case are annotated with the verdicts pass, fail and inconclusive. Inconclusive verdicts denote that neither a pass nor a fail verdict has been reached but the implementation has chosen a trace that is not included in the traces selected by the test purpose.

Although test purposes are complete, i.e. they allow actions in each state, the derived test cases satisfy Definition 3.10. That is, a test case either provides a stimulus to the implementation or it accepts all possible responses from the implementation under test.

Given a test purpose and a formal specification, TGV generates either a single test case or a complete test graph, which contains all test cases corresponding to the test purpose.

Definition 6.2 (Complete Test Graph) *A complete test graph is a deterministic LTS $CTG = (Q, L_I \cup L_U, \rightarrow, q_0)$ which has three sets of trap states Pass $\subset Q$, Fail $\subset Q$, and Inconclusive $\subset Q$ characterizing verdicts. A complete test graph satisfies the following properties:*

1. *CTG mirrors the input and output actions of the specification and considers all possible outputs of the IUT.*
2. *From each state a verdict must be reachable.*
3. *States in Fail and Inconclusive are only directly reachable by inputs.*
4. *A complete test graph is input complete in all states where an input is possible, i.e., $\forall q \in Q \bullet ((\exists a \in L_I : q \xrightarrow{a}) \Rightarrow \forall b \in L_I : q \xrightarrow{b}))$.*

As a major strength of TGV, the test case synthesis is conducted on-the-fly: parts of *S*, *SP*, and SP^{VIS} are constructed only when needed. In practice, this allows one to apply TGV to large specifications.

6.2 Test Purposes in UTP

In the context of refinement test purpose based test case selection can be described by the following statement. Given a test purpose TP, a test case TC and a specification S, then we have

$$TP \sqsubseteq_{ioco} TC \sqsubseteq_{ioco} S$$

That is, a test purpose is more abstract (with respect to $\sqsubseteq_{ioco}$) than a test case. A test case itself can be seen as an abstract specification according to $\sqsubseteq_{ioco}$. Basically, test cases are abstractions of specifications. Note that this is only true if $\sqsubseteq_{ioco}$ is used as refinement relation. When refinement ($\sqsubseteq$) is used then this does not hold. According to the definition of $\sqsubseteq_{ioco}$ S may behave arbitrary after unspecified inputs of TC.

As already mentioned we present three different approaches for test purpose based test case selection. Our first approach selects not only one test case per test purpose but multiple test cases for a single test purpose. More formally, given a test purpose TP we generate a set of n test cases $TC_1, \ldots, TC_n$ such that

$$TP \sqsubseteq_{ioco} \bigsqcap_{i=1,\ldots,n} TC_i \sqsubseteq_{ioco} S$$

The second test purpose selection approach is based on mutation based testing (Aichernig and Corrales Delgado, 2006). The idea behind mutation based test purpose selection is to generate test purposes, which lead to test cases that reveal implementations implementing faulty (mutated) specifications. Let S be the original specification and let S^m be a mutant derived from S, then a test purpose TP based on mutation testing satisfies the following statement

$$TP \sqsubseteq_{ioco} S \wedge TP \not\sqsubseteq_{ioco} S^m$$

As illustrated in Section 6.4.2, a mutation leads to a useful test case only if the mutated specification S^m does not conform to the original specification S, i.e. $S \not\sqsubseteq_{ioco} S^m$.

Finally, we consider coverage based test purposes. Coverage based test purposes lead to test cases that cover important aspects of a high level specification. Formally, this is similar to mutation based testing. As shown by Offutt and Voas (1996) some coverage criteria are subsumed by mutation testing techniques. Thus, let TP be a test purpose and let S^c be a specification that represents a mutant related to a particular coverage item, then we have

$$TP \sqsubseteq_{ioco} S \wedge TP \not\sqsubseteq_{ioco} S^c$$

6.3 Multiple Test Cases per Test Purpose

Usually, given a test purpose, tools like TGV can derive either a single test case satisfying the test purpose, or a graph that subsumes all possible test cases. To create good test suites with the former strategy, a set of suitable test purposes is required. Creating several test cases for each test purpose therefore seems to be a feasible alternative. However, the generation of a graph that subsumes all test cases as provided by TGV cannot be seen as a final step in the test case generation. Potentially, such a graph might represent an infinite number of test cases. Unfortunately, not every linear trace of the graph is a valid test case according to the theory of input-output conformance testing.

There is a whole spectrum of different possibilities for test case selection ranging from a single test case up to all possible test cases for a given test purpose. Surprisingly, the issue of whether and how to select several test cases for a test purpose has hardly been considered before. For this we use established test selection strategies for models based on coverage criteria. We extend observer automata based techniques (Blom et al., 2004; Hessel and Pettersson, 2007) to test case selection to input-output conformance testing theory, and show how this can be applied to select several test cases from a single test purpose.

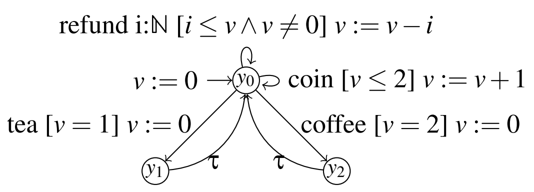

refund i:ℕ [$i \leq v \wedge v \neq 0$] $v := v - i$
$v := 0$
y_0
coin [$v \leq 2$] $v := v + 1$
tea [$v = 1$] $v := 0$
coffee [$v = 2$] $v := 0$
y_1
τ
τ
y_2

Figure 6.2: Example of a symbolic transition system representing a vending machine.

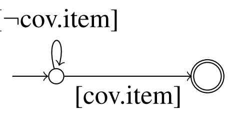


Figure 6.3: Coverage observer automaton.

The use of observer automata for generating test cases with respect to specific coverage criteria has been proposed by Blom et al. (2004). Observer automata can be created for many different coverage criteria. They are used during state space exploration in order to detect when coverage items have been reached and return corresponding test cases. Informally, the superposition of an observer automaton and the model is calculated and traversed. Whenever an observer enters an accepting state the linear trace that lead to this state is a test case that covers the coverage item represented by the observer.

Blom et al. (2004) use extended finite state machines (EFSMs) for representing observer automata. As we apply observer automata to labeled transition systems our observer automata are given in terms of symbolic transition systems (STSs, (Frantzen et al., 2004)).

Definition 6.3 (Symbolic transition system) *A symbolic transition system is a tuple $\mathcal{S} = (L, l_0, \mathcal{V}, I, \Lambda, \rightarrow)$, where L is a set of locations and $l_0 \in L$ is the initial location. $\mathcal{V}$ is a set of location variables and I is a set of interaction variables with $\mathcal{V} \cap I = \emptyset$. Λ is the set of actions, and $\tau \notin \Lambda$ denotes an unobservable action. $\rightarrow$ denotes the transition relation, where each element $(l, \lambda, \phi, \rho, l') \in \rightarrow$, has a source location l, a target location l', a possibly parametrized action λ, a guard ϕ and a update function* ρ.

Example 6.2. Figure 6.2 shows an example STS, representing a drink vending machine. The locations L are given by $\{y_0, y_1, y_2\}$ with y_0 as initial location. We have one location variable, i.e. $\mathcal{V} = \{v\}$ and one interaction variable $I = \{i\}$. The set of actions Λ is given by $\{\text{refund}, \text{tea}, \text{coin}, \text{coffee}\}$. Transitions are labeled with an action, guard conditions are included in brackets [] and followed by the update function. The variable v stores the number of inserted coins. The action refund uses an additional parameter i, which is a natural number. Due to the guard this parameter is less than or equal to the number of inserted coins, i.e., $i \leq v$. Thus, this action allows the rejection of the inserted money. □

An observer automaton is a symbolic transition system that is parametrized by a particular coverage item.

Example 6.3. Figure 6.3 depicts an observer automaton as an STS. The observer stays in its initial state as long as the desired coverage item is not reached. We denote the coverage item as *cov.item*, and it can represent any entity of the LTS that should be covered, e.g. states, labels, or transitions. Formally, symbolic transition systems do not provide accepting states as required by observer automata, therefore we use sink states, i.e., states without outgoing transitions, to implement accepting states. □

Because test cases need to satisfy additional properties (Definition 3.10) that cannot always be fulfilled by linear traces a test case is an LTS which has to be input complete in states where inputs are possible and controllable. In addition, the definition of a test case requires that only responses from the system under

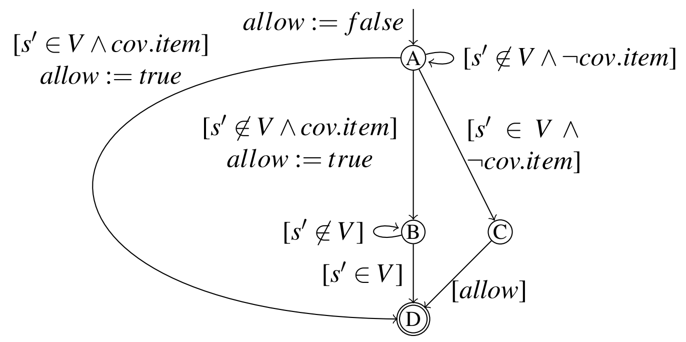

$allow := false$
$[s' \in V \wedge cov.item]$
$allow := true$
$[s' \notin V \wedge \neg cov.item]$
$[s' \notin V \wedge cov.item]$
$allow := true$
$[s' \in V \wedge \neg cov.item]$
$[s' \notin V]$
$[s' \in V]$
$[allow]$
A
B
C
D

Figure 6.4: Extended observer for the generation of ioco-correct test cases.

test lead to the verdicts *fail* and *inconclusive*, and that from each state a verdict must be reachable. Finally, outputs of the system under test have to be inputs for the test case and vice versa.

Except for controllability, a test graph generated by TGV fulfills all the required properties. Thus, any approach that extracts test cases from such a test graph needs to preserve these properties, while it must additionally ensure controllability. Below we show how to extend the approach based on observer automata in order to generate valid test cases with respect to Definition 3.10.

6.3.1 Extending observer automata

In its original definition (Blom et al., 2004), an observer automaton generates a linear test case that ends as soon as the coverage item has been covered. Such linear traces do not satisfy the requirements of our test cases.

Because test cases have to be input complete (Definition 3.10) they might include traces that do not cover the observer's coverage item. Such traces result from inputs that have to be selected in order to satisfy input completeness. When superposing the observer and the complete test graph the observer can be in different states after different traces. Hence, an extended observer keeps track of multiple traces.

However, observer automata have to ensure that every trace of a generated test case ends in a verdict state. Therefore, we introduce an additional variable *allows* that is used to take care of traces leading to verdict states but not covering the coverage item of the observer. Such traces are only allowed for a test case if the coverage item has been covered by another trace selected by the observer.

This extension is illustrated in Figure 6.4. Any observer automaton for a particular coverage item (cov.item) can be extended in this way. Let $V = Pass \cup Fail \cup Inconclusive$, then for an edge (s,a,s') of the superposition of the observer automaton and the model this extension distinguishes four different cases:

1. The edge leads to a verdict state and the coverage item is met by that edge: $s' \in V \wedge cov.item$
2. The edge meets the coverage item but it does not lead to a verdict state: $s' \notin V \wedge cov.item$
3. The edge leads to an verdict state, but the edge is not related to the coverage item: $s' \in V \wedge \neg cov.item$
4. The edge does not lead to a verdict state and the coverage item is not met: $s' \notin V \wedge \neg cov.item$

6.3.2 Ioco based test case extraction

Algorithm 6.1 depicts the proposed procedure that deals with the discussed issues. Given a complete test graph (CTG) and a set of observers, the basic idea of the algorithm can be summarized as follows: Starting at the initial state of the CTG and the initial states C of the observers we traverse through the CTG. For each state of the complete test graph we keep track of the corresponding observer states. Furthermore, we store for each state the path that leads to this state. If an observer reaches its accepting state this stored path corresponds to a test case that covers the observers coverage item.

When processing the edges of a particular CTG state we distinguish between inputs and outputs. For input actions we copy all input actions enabled in the considered state to the generated test case and take care that the observer keeps track of all successor states. For every output edge we add the output edge to the test case. If there are both inputs and outputs enabled in a particular state we follow the paths for inputs and the paths for every output separately. Note that in this case we have to use a different *allow* variable for each path.

In detail, the algorithm uses two data structures *wait* and *pass* for maintaining states waiting to be examined and states already examined, similar to the algorithm of Hessel and Pettersson (2007). The set *wait* consists of sets of triples $\langle s, C, t\rangle$ and the test case ω associated with these triples. Each triple comprises the state of the test graph s, the states of the observers C and the corresponding state of the generated test case t.

As long as there are elements in *wait* (Line 2), the algorithm takes pairs (P, ω) from *wait*. Then it iterates over all triples in P (Line 5) and considers the successor states of $\langle s, C\rangle$ given by the transition relation of the observer/test graph superposition (Line 6 and 20). If a successor state is reached by an output action (Line 6) then the new set of triples P' is calculated by taking the triples of P (Line 8). The selected triple $\langle s, C, t\rangle$ is replaced within P' by a new triple built from the successor $\langle s', C'\rangle$ of $\langle s, C\rangle$ and a new state t_{new} within the partially built test case ω. If this state P' has not been processed previously (Line 9) then the test case ω is updated by adding the considered transition to ω (Line 10). Note that "_" in Line 9 and in Line 25 matches any test case state, achieving that the conditions in these lines only check whether the combination of the current observer state and the current state in the complete test graph are neither in *pass* nor *wait*. If all triples in P' correspond to final states in the observer (Line 11), then the test case is added to the test suite, otherwise the new P' is added to *wait*.

If a successor is connected via an input edge (Line 20), then the successor triple is added to a temporary data-structure P' that will hold all successor triples reachable via inputs. After iterating over all successors (Line 20) the test case is either added to the test suite (Line 27) or the newly generated state is added to *wait* (Line 29).

Optimization As we are interested in a set of test cases such that every observer is covered, the algorithm illustrated in Figure 6.1 can be further optimized such that the number of generated test cases is reduced. Similar to Hessel and Pettersson (2007) we approximate the set of test cases by adding a test case (Lines 12 and 27) only to the test suite if it covers a currently uncovered observer.

Restrictions In some cases loops in the complete test graph can lead to problems when trying to extract a test case. If a loop in the complete test graph starts with an input edge and there are other input edges but no output edges enabled in that state we fail to extract a test case. This is because when the state is revisited we need to follow all input edges again (if a test case accepts an input, it accepts all inputs, see Definition 3.10). An observer will never reach an accepting state on the looping edge since that trace does not end in an verdict state. Note that two triples $\langle s, C, t\rangle$ and $\langle s', C', t'\rangle$ are equal if and only if $s = s' \wedge C = C'$. We do not consider the state of the test case here, otherwise our algorithm would not terminate for the case described above.

This restriction is not a problem in general. Whenever the setting described above occurs in an complete test graph we may rewrite the corresponding test purpose such that there is no loop. For example, refusing or accepting the looping edge after the path leading to the loop will be sufficient.

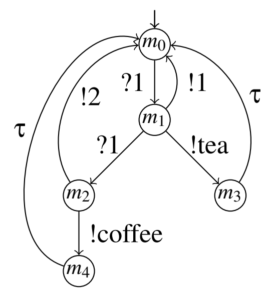
Algorithm 6.1 Extended observer automata based test case extraction algorithm.

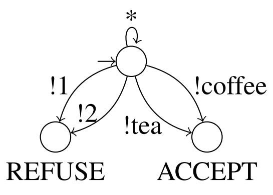
```
1: $pass \leftarrow \emptyset$; $wait \leftarrow \{(\langle s_0, C_0, t_0\rangle, \varepsilon)\}$
2: while $wait \neq \emptyset$ do
3:     select $(P, \omega)$ from $wait$
4:     add $(P, \omega)$ to $pass$
5:     for all $\langle s, C, t\rangle \in P$ do
6:         for all output edges $e : \langle s, C\rangle \xrightarrow{a} \langle s`, C`\rangle$ do
7:             $t_{new} \leftarrow$ new (unused) state in $\omega$
8:             $P' \leftarrow P \backslash \langle s, C, t\rangle \cup \langle s`, C`, t_{new}\rangle$
9:             if $(P', \_) \notin (pass \cup wait)$ then
10:                $\omega' \leftarrow \omega \cup (t, b, t_{new})$
11:                if new covered observer in $C'$ then
12:                    add $\omega'$ to test suite
13:                else
14:                    add $(P', \omega')$ to $wait$
15:                end if
16:            end if
17:        end for
18:        if there are input edges then
19:            $(P', \omega') \leftarrow (P, \omega)$
20:            for all input edges $\langle s, C\rangle \xrightarrow{b} \langle s`, C`\rangle$ do
21:                $t_{new} \leftarrow$ new (unused) state in $\omega$
22:                $P' \leftarrow P' \backslash \langle s, C, t\rangle \cup \langle s`, C`, t_{new}\rangle$
23:                $\omega' \leftarrow \omega' \cup (t, b, t_{new})$
24:            end for
25:            if $(P', \_) \notin (pass \cup wait)$ then
26:                if new covered observer in $C'$ then
27:                    add $\omega'$ to test suite
28:                else
29:                    add $(P', \omega')$ to $wait$
30:                end if
31:            end if
32:        end if
33:    end for
34: end while
```

6.3.3 Test case extraction: An example

The labeled transition systems shown in Figure 6.5 and in Figure 6.6 serve to illustrate the presented algorithm. Assume we have the test purpose depicted in Figure 6.5b and a single observer automaton with the aim to cover the label *?coffee*. The observer automaton basically looks like Figure 6.4, and uses $label = ?coffee$ as coverage item *cov.item*. Using the test purpose on the specification shown in Figure 6.5a leads to the complete test graph of Figure 6.5c. Superposing the observer automaton on this complete test graph leads to the graph shown in Figure 6.6a. For the sake of clarity, the superposition does not include the test case states and the test cases itself. Finally, Figure 6.6b shows the test case that will be generated in our example. Recall that outputs of a test graph are stimuli to be sent to an implementation and thus inputs in terms of an specification. In contrast, inputs of a CTG are outputs of the implementation. The same is true for test cases. However, test purposes operate on the specification and therefore use the specification's notion of inputs and outputs.

In detail, Algorithm 6.1 works as follows: The algorithm starts with the initial states of the complete test graph and of the observer automaton, i.e., $wait \leftarrow \{(\langle 1, A, t_0\rangle)\}$ (Line 1). It then takes this set of triples (s_0 of Figure 6.6a) and processes one triple after the other.





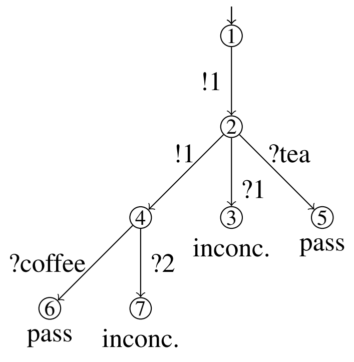
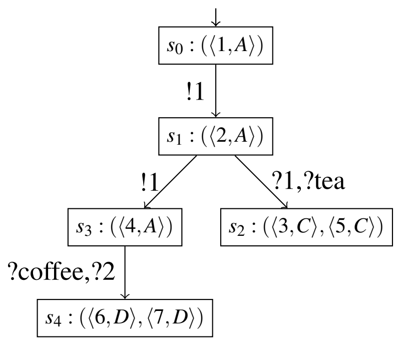


Figure 6.5: Examples of an input-output labeled transition system (a), a test purpose (b) and the resulting complete test graph (c).


$s_0 : (\langle 1,A \rangle)$
!1
$s_1 : (\langle 2,A \rangle)$
!1
?1,?tea
$s_3 : (\langle 4,A \rangle)$
$s_2 : (\langle 3,C \rangle, \langle 5,C \rangle)$
?coffee,?2
$s_4 : (\langle 6,D \rangle, \langle 7,D \rangle)$
(a)

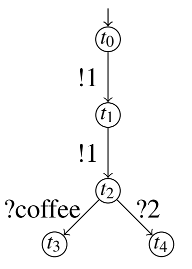


Figure 6.6: Superposition of an observer and a complete test graph (a), and a test case (b).

Since the first and only triple of s_0 is $\langle 1, A, t_0 \rangle$, the algorithm considers the output edges enabled in state 1 of the complete test graph. The only edge in 1 is !1 leading to a new state in the superposition, i.e., $s_1 = (\langle 2, A \rangle)$ (Line 6). Since s_1 is neither in *pass* nor in *wait* the considered edge !1 is added to ω'. As the observer is not covered yet we add the new state s_1 and the corresponding test case (comprising only one edge at the moment) to the set of waiting states (Line 14).

The algorithm continues at Line 3. It picks s_1 from *wait*, adds s_1 to the set *pass* and processes the only triple $\langle 2, A, t_1 \rangle$ of s_1. In state 2 of the CTG we have both, input and output edges. The algorithm considers the output edge first: The new set of triples s_3 representing the successor state of s_1 in the superposition (Line 8) is calculated. Because s_3 is neither in *pass* nor in *wait*, a new test case ω' extended by $t \xrightarrow{!1} t_{new}$ is generated. The observer is still in state A, thus (P', ω') is added to *wait*.

The two input edges labeled with ?1 and ?*tea* in state 2 of the complete test graph lead to s_2 of the superposition. The test case ω' associated with s_2 comprises three edges: $t_1 \xrightarrow{!1} t_2$, $t_2 \xrightarrow{?1} t_3$, and $t_2 \xrightarrow{?tea} t_4$. However, the observer is not covered by this test case because the observer is in state C for both triples of s_2 and C is not the accepting state of the observer. Thus, s_3 and ω' are added to *wait* (Line 14).

The algorithm picks the next waiting state from the set *wait*. Assume that s_2 is picked next. The algorithm iterates over all triples of s_2, i.e., $\langle 3, C \rangle$ and $\langle 5, C \rangle$. Therefore, it considers the states 3 and 5 of the complete test graph which both have no outgoing edges. Consequently, s_2 is moved to *pass* only (Line 4).

Taking the next (P, ω) tuple from *wait* gives $P = (\langle 4, A \rangle)$ and a test case ω that looks like $t_0 \xrightarrow{!1} t_1 \xrightarrow{!1} t_2$.

By processing the triples of P the algorithm considers state 4 of the complete test graph.

Since 4 has only input edges the algorithm continues in Line 18. After creating the temporary set of triples P' and the temporary test case ω' (Line 19) the algorithm processes the two edges $4 \xrightarrow{?coffee} 6$ and $4 \xrightarrow{?2} 7$ and updates P' and ω'. Because the *allows* variable of the observer is set to true when processing $4 \xrightarrow{?coffee} 6$, we can move from the observer state C for the edge $4 \xrightarrow{?2} 7$ to the observer state D. Thus, the algorithm comes up with s_4 comprising $\langle 6, D\rangle$ and $\langle 7, D\rangle$. The test case associated with that state is shown in Figure 6.6b.

The state s_4 is neither in *pass* nor in *wait* (Line 25). In the next step the algorithm checks if there are new covered observers. The generated test case covers the observer since the observer state in both triples is the accepting state D. Thus, the test case is added to the test suite. Finally, the algorithm continues at Line 2 where it notices that the set of waiting states is empty and thus the algorithm terminates.

Please note that edges leading to fail verdicts are not explicitly existent in complete test graphs, so are they in our test cases. Every input edge not enabled in a state where inputs are possible implicitly leads to a fail verdict.

6.3.4 Correctness and runtime analysis

In order to show the correctness of our extended test case extraction algorithm we need to answer the question of termination first. For termination, we need to analyze the number of states of the superposition of the observer automata and the complete test graph.

Lemma 6.1 (Finite Superposition) *The number of states of the superposition of observers and a complete test graph is finite.*

Theorem 6.1 *Algorithm 6.1 terminates and has an asymptotic runtime of $O((2^n * |Q|)^{(2^n * |Q|)})$ where $|Q|$ is the number of states in the complete test graph and n is the number of observers.*

Proof 6.1. Every state of the superposition of observers and a complete test graph is either in *pass*, in *wait* or has not been visited yet. Newly explored states are added to *wait* only if they are not yet in the set of waiting states and if they have not been processed yet, i.e. they are not in *pass*. If a state is processed it is moved from the set of waiting states to the set *pass*. Thus, each state is only visited once. Since the superposition of observers and a complete test graph has an upper bound of $O((2^n * |Q|)^{(2^n * |Q|)})$ (Lemma 6.1) and each state is visited once, Algorithm 6.1 terminates with a worst case asymptotic runtime of $O((2^n * |Q|)^{(2^n * |Q|)})$. □

The runtime estimation for our algorithm is a worst case approximation. It is still an open issue whether this is the smallest upper bound. We believe that there is a smaller upper bound since we have considered only the number of possible combinations of tuples. Many of these tuples may not be possible. For example, a state of a complete test graph can only be combined with two different observer states if the state is revisited, i.e., if it is included in a loop. This is supported by our experimental results, which suggest that our algorithm terminates for practical problems within reasonable time.

Lemma 6.2 (Reachable verdict) *A verdict state is reachable from each state of a generated test case.*

Lemma 6.3 (Input enabled) *The generated test cases are input complete in all states where inputs are possible.*

Lemma 6.4 (Controllability) *Every generated test case is controllable.*

Theorem 6.2 *Algorithm 6.1 generates valid test cases with respect to Definition 3.10.*

Proof 6.2. A test case TC is valid with respect to Definition 3.10 if

1. TC mirrors the input and output actions of the specification and considers all possible outputs of the IUT
2. From each state a verdict must be reachable
3. States in *Fail* and *Inconclusive* are only directly reachable by inputs
4. A test case is input complete in all states where an input is possible
5. TC is controllable, i.e., no choice between two outputs or between inputs and outputs.

By lemma 6.2, 6.3, and 6.4 the generated test cases satisfy the properties 2, 4, and 5. Assume that either property 1 or property 3 does not hold because we do not add new edges but only use edges from the complete test graph. This means that these properties do not hold on the CTG either. This contradicts with the definition of the CTG. Thus, the generated test cases are valid with respect to Definition 3.10. □

6.3.5 Coverage Criteria for Labeled Transition Systems

Observer automata can represent various coverage criteria. For our purpose we use five different coverage criteria on the complete test graph: state coverage, input label coverage, output label coverage, all label coverage, and transition coverage.

Having different coverage criteria on the types of labels is inspired by the properties of the **ioco** relation. That is, the outputs of the IUT (i.e., inputs to the test case) are more important than stimuli because stimuli do not directly lead to errors (see Property 3 of Definition 6.2).

Definition 6.4 (State Coverage) *A state $q \in Q$ of a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ is covered by test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, if $q \in Q^t$.*

The state coverage value is calculated as the ratio of covered states to states in total in the LTS. Note that we cannot cover states reachable by unobservable (τ) actions only, because test cases do not allow for unobservable actions.

Definition 6.5 (Input Label Coverage) *An input label $l_I \in L_I$ of a labeled transition system $M = (Q, L_I \cup L_U \cup \{\tau\}, \rightarrow, q_0)$ is covered by test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, if $l_I \in L^t$ and there exists a transition $(q, l_I, q') \in \rightarrow_t$ reachable from the initial state q_0^t.*

The input label coverage value is obtained by dividing the number of covered input labels by the number of all input labels in the LTS. Input label coverage applied to a complete test graph gives the coverage of stimuli to be sent to an implementation.

Definition 6.6 (Output Label Coverage) *An output label $l_U \in L_U$ of a labeled transition system $M = (Q, L_I \cup L_U \cup \{\tau\}, \rightarrow, q_0)$ is covered by test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, if $l_U \in L^t$ and there exists a transition $(q_1, l_U, q_2) \in \rightarrow_t$, which is reachable from the initial state q_0^t.*

The output label coverage value is given as the ratio of covered output labels to all output labels in the LTS. Output label coverage applied to a complete test graph gives the coverage of responses possibly received from an implementation.

Definition 6.7 (Label Coverage) *A label $l \in L$ of a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ is covered by test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, if $l \in L^t$ and there exists a transition $(q_1, l, q_2) \in \rightarrow_t$ reachable from the initial state q_0^t.*

The label coverage represents the percentage of labels of the LTS that are covered. Obviously, the τ action cannot be covered by a test case, since this action is unobservable during testing it cannot occur in a test case.

Definition 6.8 (Transition Coverage) *A transition $(q,l,q') \in \rightarrow$ of a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, where $q,q' \in Q$ and $l \in L$, is covered by test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, if $(q,l,q') \in \rightarrow_t$ and (q,l,q') is reachable from the test case's initial state q_0^t.*

The transition coverage represents the percentage of transitions of the LTS that are covered.

Note that in the case of a complete test graph, transition coverage subsumes label coverage and state coverage. This is because there are no states unreachable from the initial state and because there are no labels that are not used on any transition. Furthermore, label coverage subsumes input label coverage and output label coverage. Note further that label coverage does not subsume state coverage, since the same label may be used on different transitions.

6.4 Mutation-based Test Purposes

Selecting multiple test cases for a single test purpose requires the existence of a proper set of test purposes. However, as there is always insufficient time for testing it is also interesting to automatically generate test purposes. One possibility for test case selection is the use of anticipated faults for the generation of test cases. This idea dates back to the late 1970s (DeMillo et al., 1978; Hamlet, 1977) where testers mutated a source code to assess their test cases. Budd and Gopal (1985) applied this technique to specifications.

A fault is modeled at the specification level by altering the specification syntactically. The idea is to generate test cases that would fail if an implementation conforms to a faulty specification. Aichernig and Corrales Delgado (2006) proposed the following procedure to generate fault-based test cases for LOTOS specifications:

- Select a LOTOS mutation operator and generate a mutated version of the LOTOS specification.
- Generate the labeled transition systems S_τ and S_τ^m for the specification and its mutated version.
- Simplify S_τ and S_τ^m with respect to the Safety Equivalence relation in order to obtain S and S^m
- Check for strong bisimulation equivalence between S and S^m.
- If the equivalence check fails, use the counter-example (extended by one transition) as a test purpose for the TGV tool.
- Run the TGV tool with the test purpose TP on the original (non-mutated) specification in order to derive the final test case.

Although Aichernig and Corrales Delgado (2006) applied this approach by testing a web-server, we observed several open issues when applying the method in an industrial project. First, the used bisimulation check over-approximates the set of needed test cases, because the generated test cases are intended for input-output conformance (**ioco**) testing with respect to the **ioco**-relation. Albeit, due to the use of TGV all generated test cases are sound, i.e. test cases do not fail on input-output conform implementations. However, there are test cases for faults that cannot be detected using the ioco-relation. Thus, using bisimulation results in more test cases than needed. Second, the original approach relies on the construction of the complete state spaces of both, the original specification and the mutated specification. For industrial specifications with huge state spaces this is often infeasible. In the following we first briefly discuss the used mutation operators and then address these two issues.

6.4.1 Mutating LOTOS Specifications

As we introduce faults at the level of the specification mutation operators are needed. These operators represent the sort of faults that we consider. For the selection of mutation operators one usually relies on two hypotheses (Budd et al., 1980). The first one is called the ‘competent specifier hypothesis’ which is related to the ‘competent programmer hypothesis’ (Budd et al., 1980). This hypothesis states that the specifier (programmer) is usually competent and gets the specification (program) almost correct. Faults can be corrected by a few key strokes.

The second assumption is called the ‘coupling hypothesis’. It states, that big and dramatic effects that arise from bugs in software are closely coupled to small and simple failures.

Based on these two assumptions we can stick to small mutations on the specification. Thus, as usual in mutation testing, we use small syntactic changes on LOTOS specifications; each mutant only comprises a single mutation.

We use some of the mutation operators proposed by Black et al. (2000) and Srivatanakul et al. (2003) and adapted them to LOTOS specifications. Our mutation operators are listed in Table 6.1. Besides, an abbreviation (column Op.), the full name and a description we list how LOTOS specifications are affected by these operators. *Data* mutations make changes to data expressions, e.g. change variables in logical expressions. On the contrary, *action* mutations directly alter LOTOS actions, e.g. change the gate of an action. *Structural* mutations affect the structure by for example replacing composition operators.

6.4.2 Fault-based IOCO Testing

We are interested in testing for input-output conformance. Thus, for every mutant we want to generate a test case such that the test case fails if this mutant has been implemented. However, not all mutation operators lead to models that can be distinguished from the original specification when using **ioco**, i.e. not all mutations represent faults. A fault can only be defined with respect to a conformance relation.

A mutant that cannot be distinguished from the original specification is called equivalent mutant. Although, the **ioco** relation is not an equivalence relation, we still use the terms equivalent and non-equivalent mutant as they are common in mutation testing. For an equivalent mutant there is no test case that distinguishes the mutant from the original specification. On the contrary, a non-equivalent mutant comprises a fault such that there is a test case that passes on the original specification and fails on the mutant.

In the following the meaning of faults in the context of **ioco** is shown. Our first observation is that not all injected faults will cause observable failures. In order to observe a failure, the mutant must not conform (with respect to **ioco**) to our original specification. Hence, given an original specification S we are only interested in mutants S^m, such that

$$S \not\sqsubseteq_{ioco} S^m$$

Unfolding the definition of $\sqsubseteq_{ioco}$ gives

$$\neg\left[\forall t \in \mathcal{A}_\delta^*, \forall o \in \mathcal{A}_{out} \bullet \left(\begin{array}{c} ((Trace(S)[t/tr'] \wedge Trace(S^m)[t\hat{\ } o/tr']) \Rightarrow Trace(S)[t\hat{\ } o/tr']) \wedge \\ ((Trace(S)[t/tr'] \wedge Quiet(S^m)[t/tr']) \Rightarrow Quiet(S)[t/tr']) \end{array}\right)\right]$$

By simply shifting the negation operator into the formula we obtain the following formula, where v denotes the vector of all observation variables, i.e. $v = \{tr, tr', ref, ref', wait, wait', ok, ok'\}$.

$$\exists v, \exists t \in \mathcal{A}_\delta^*, \exists o \in \mathcal{A}_{out} \bullet \left(\begin{array}{c} (Trace(S)[t/tr'] \wedge Trace(S^m)[t\hat{\ } o/tr'] \wedge \neg(Trace(S)[t\hat{\ } o/tr'])) \vee \\ (Trace(S)[t/tr'] \wedge Quiet(S^m)[t/tr'] \wedge \neg(Quiet(S)[t/tr'])) \end{array}\right)$$

Table 6.1: Mutation operators for LOTOS specifications.

Op.	Name	Type	Description
ASO	Association Shift Op.	data	Change the association between variables in boolean expressions, e.g. replace $x \wedge (y \vee z)$ by $(x \wedge y) \vee z$.
CRO	Channel Replacement Op.	action	Replace the communication channel, i.e. change the gate of an event. For example, this operator would change Line 19 of Figure 3.11 from `out !error` to `ui !error`.
EDO	Event Drop Op.	action	Drop events of the specification.
EIO	Event Insert Op.	action	Duplicate existing events.
ENO	Expression Negation Op.	data	Replace an expression by its negation, e.g. replace $x \wedge y$ by $\neg(x \wedge y)$.
ERO	Event Replacement Op.	action	Replace an event by a different event. For example, applying this operator to Line 19 of Figure 3.11 leads to `out !error` to `out`.
ESO	Event Swap Op.	action	Swap two neighbouring events.
HDO	Hiding Delete Op.	action	Delete an event from hide definition, i.e. make an unobservable event observable.
LRO	Logical Operator Replacement	data	Replace a logical operator by other logical operators.
MCO	Missing Condition Op.	data	Delete conditions from decisions.
ORO	Operand Replacement Op.	data	Replace an operand (variable or constant) by another syntactically legal operand, e.g. one mutation with respect to this operator is to replace the stack variable s in Line 19 of Figure 3.11 by `nil`.
POR	Process Operator Replacement	structure	Replace synchronization operators (||,|[...]|,|||).
PRO	Process Replacement Op.	structure	Replace process instantiations with stop or exit events.
RRO	Relational Operator Replacement	data	Replace a relational operator ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) by any other except its opposite (since the opposite is similar to the negation operator).
SNO	Simple Expression Negation	data	Replace a simple expression by its negation, e.g. negate x in $x \wedge y$ getting $(\neg x \wedge y)$.
SOR	Sequential Operator Replacement	structure	Replace the sequential composition operator >> and [>.
USO	Unobservable Sequence Op.	action	Make events of the specification unobservable.

This can further be rewritten to

$$\exists v, \exists t \in \mathcal{A}_\delta^* \bullet \left(Trace(S)[t/tr'] \wedge \left(\begin{array}{c} (\exists o \in \mathcal{A}_{out} \bullet (Trace(S^m)[t\hat{\ } o/tr'] \wedge \neg(Trace(S)[t\hat{\ } o/tr']))) \vee \\ (Quiet(S^m)[t/tr'] \wedge \neg(Quiet(S)[t/tr'])) \end{array} \right) \right)$$

This is the first hint for a testing strategy. We are interested in the suspension trace of actions, i.e. $\exists t \in \mathcal{A}_\delta^*$ such that t is a valid trace of the specification, leading to non-conformance between the mutant and the original specification. In other words, our test purposes for detecting faults are sequences of actions leading to non-conformance.

As this formula shows, non-conformance is given if there is an output in the mutant ($Trace(S^m)[t\hat{\ } o/tr']$) but this output is not allowed by the specification ($\neg(Trace(S)[t\hat{\ } o/tr'])$), or if the mutant is quiescent ($Quiet(S^m)[t/tr']$) but the specification does not allow quiescence ($\neg(Quiet(S)[t/tr'])$).

This simple derivation shows an important property of the **ioco** relation: mutating the specification by injecting an additional input a such that a new trace for the mutant S^m is generated, i.e. $\forall t \in \mathcal{A}_\delta^* \bullet \neg(Trace(S)[t\hat{\ } \langle a \rangle /tr'])$, does not lead to a failure. Furthermore, removing an output from S to get the mutant S^m, such that quiescence is not effected, does also not induce an observable difference.

The theory highlights a further important clarification in fault-based testing: In the presence of non-determinism, there is no guarantee that an actual fault will always be detected. Non-conformance only means that there is a wrong output, i.e. $\exists o \in \mathcal{A}_{out} \ldots$, after a trace of actions, but the implementation may still opt for a correct one. In that case we rely on the complete testing assumption (Luo et al., 1994), which says that an implementation exercises all possible execution paths of a test case t, when t is applied a predetermined finite number of times.

6.4.3 On-the-fly IOCO Checking

As highlighted above we only need to consider mutants S^m of a specification S such that $\neg(S^m$ **ioco** $S)$. Because the state spaces of specifications are usually huge, we cannot construct the state space of the specification and the mutant in advance, and then check for conformance. Thus, conformance checking between the mutant and the specification has to be done on the fly.

Therefore, we use the LOTOS parser of the CADP toolbox (Garavel et al., 2002). This parser takes a LOTOS specification and allows one to access the underlying LTS incrementally. The LOTOS specification is translated into an initial-state and a successor-function. The successor-function takes a state and returns the edges, i.e. labels and end-states, that are enabled in the given state.

In order to check two labeled transition systems for conformance we use an approach similar to the approach of Fernandez and Mounier (1991). That is, we define a synchronous product ($\times_{ioco}$) between two labeled transition systems S^m and S such that $S^m \times_{ioco} S$ contains special fail states if $\neg(S^m$ **ioco** $S)$. Checking for conformance is then implemented as a simple reachability search for fail states. If there is a fail state after a particular path, then this path is a counter-example showing the non-conformance between S^m and S.

Since the input-output conformance relation uses δ-labeled transitions and these transitions are not initially provided by the semantics of LOTOS specification we have to identify and to label quiescent states before calculating the synchronous product. More precisely, we add quiescence labeled transitions for quiescent states when iterating over the transitions of a particular state.

After adding the quiescence information we make the two labeled transition systems deterministic. This is done during the calculation of the synchronous product by the use of the subset construction (Hopcroft and Ullman, 1979). Note that in the worst case this may cause an exponential increase of the number of states. During the process of making the LTSs deterministic we remove τ-labeled transitions too.

Definition 6.9 *Let $S^m = (Q^{S^m}, L \cup \{\tau,\delta\}, \rightarrow_{S^m}, q_0^{S^m})$ and $S = (Q^S, L \cup \{\tau,\delta\}, \rightarrow_S, q_0^S)$ be two deterministic LTSs, where the labels L are partitioned into inputs L_I and outputs L_U, i.e. $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$. The synchronous product $SP = S^m \times_{ioco} S$ is an LTS $SP = (Q^{SP}, L, \rightarrow_{SP}, q_0^{SP})$, where its state set Q^{SP} is a subset of $(Q^{S^m} \times Q^S) \cup \{pass, fail\}$ reachable from the initial state $q_0^{SP} =_{df} \left(q_0^{S^m}, q_0^S\right)$ by the transition relation $\rightarrow_{SP}$. Let $q^{S^m} \in Q^{S^m}$ and $q^S \in Q^S$ be two states of S^m and S, such that $\left(q^{S^m}, q^S\right) \in Q^{SP}$. Then, the transition relation $\rightarrow_{SP}$ is defined as the smallest set obtained by the application of the following rules:*

1. *Edges possible in both LTSs, S^m and S:*
   $\forall a \in L_I \cup L_U \bullet \forall q'^{S^m} \in Q^{S^m}, q'^S \in Q^S \bullet q^{S^m} \xrightarrow{a}_{S^m} q'^{S^m} \wedge q^S \xrightarrow{a}_S q'^S \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{a}_{SP} \left(q'^{S^m}, q'^S\right)$.
2. *Implementation freedom on unspecified inputs:*
   $\forall a \in L_I \bullet \forall q'^{S^m} \in Q^{S^m} \bullet q^{S^m} \xrightarrow{a}_{S^m} q'^{S^m} \wedge q^S \not\xrightarrow{a}_S \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{a}_{SP} pass$.
3. *S^m may allow fewer outputs than S:*
   $\forall b \in L_U \bullet \forall q'^S \in Q^S \bullet q^S \xrightarrow{b}_S q'^S \wedge q^{S^m} \not\xrightarrow{b}_{S^m} \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{b}_{SP} pass$.
4. *Input enabledness of S^m:*
   $\forall a \in L_I \bullet \forall q'^S \in Q^S \bullet q^S \xrightarrow{a}_S q'^S \wedge q^{S^m} \not\xrightarrow{a}_{S^m} \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{a}_{SP} \left(q^{S^m}, q'^S\right)$.
5. *Unspecified outputs of S^m:*
   $\forall b \in L_U \bullet \forall q'^{S^m} \in Q^{S^m} \bullet q^{S^m} \xrightarrow{b}_{S^m} q'^{S^m} \wedge q^S \not\xrightarrow{b}_S \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{b}_{SP} fail$.

Rule 1 states that edges that are possible in both LTSs are edges of the synchronous product.

Rule 2 handles the cases where the LTS representing the implementation allows inputs, that are not specified by the LTS representing the specification. Since **ioco** allows any behavior on unspecified inputs we add a pass state to the synchronous product. The added state is a sink state, i.e., there are no outgoing edges. Note that pass states do not affect the final comparison result, since only the existence of fail states determine whether two LTS are related under conformance (with respect to **ioco**) or not.

Since **ioco** requires that the outputs of the implementation's LTS have to be a subset or have to be equal to the outputs of the specification's LTS we add an edge to a pass state for any output that is allowed in S but not in S^m (Rule 3). Again, this pass state has no influence on the final comparison result.

Note that **ioco** requires the left hand side LTS to be weakly input enabled. In practice this may not be the case for a given input output labeled transition system. Thus, we have to convert the left hand side LTS to a weakly input enabled LTS. The synchronous product considers this requirement by Rule 4 of the transition relation. This rule assumes that an input is always possible in S^m. Thus, if input a is not allowed in S^m, the input enabledness allows to assume a self-loop labeled with a on state q^{S^m}. Hence, this rule ensures that the synchronous product will not contain an edge leading to fail because S^m lacks input enabledness.

We add an edge leading to a fail state if an output of the left hand LTS S^m is not an output of the right hand LTS S (Rule 5). Only in that case the two LTSs do not conform with respect to **ioco**. As the synchronous products are constructed on-the-fly we stop the construction if a fail-state is reached.

Example 6.4. Figure 6.8 shows the synchronous products used to check conformance on some of the labeled transition systems of Figure 3.2. The basis of the **ioco** check are δ-annotated labeled transition systems, which are deterministic.

When checking f **ioco** e the synchronous product according to Definition 6.9 looks like the LTS shown in Figure 6.8a. Note that f and e are already deterministic and thus the determinisation does not change these LTSs. The trace $\langle ?2 \rangle$ of the implementation's model f is not relevant in specification e, and hence Rule 2 of Definition 6.9 applies.

When checking g **ioco** h and h **ioco** g the two labeled transition systems are turned deterministic first. The resulting LTSs g' and h' are illustrated in Figure 6.7. The synchronous product using $\times_{ioco}$ is illustrated in Figure 6.8b.

The fail state comes from Rule 5 which says, that outputs of the implementation that are not allowed by the specification lead to fail. However, when checking g **ioco** h there would be a pass state instead of the

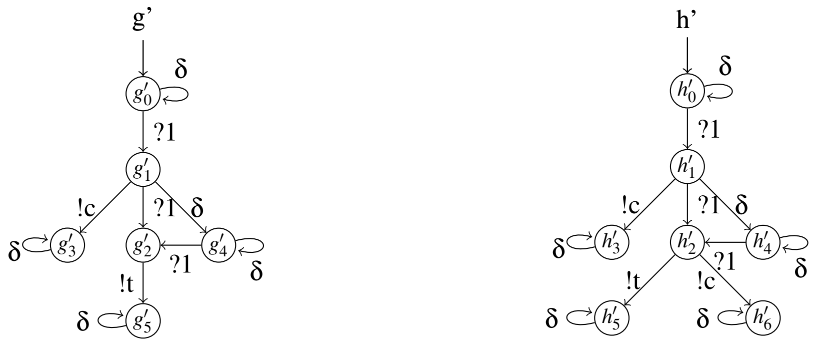
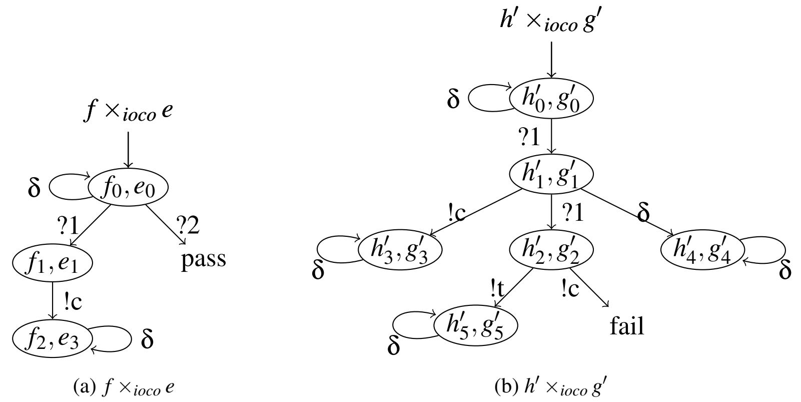


Figure 6.7: Deterministic versions of the labeled transition systems g and h of Figure 3.2.



(a) $f \times_{ioco} e$

(b) $h' \times_{ioco} g'$

Figure 6.8: Synchronous products illustrating the application of Definition 6.9 to the labeled transition systems f and e of Figure 3.2 and g' and h' of Figure 6.7.

fail state. This is because implementations may have fewer outputs than specifications after a particular trace (Rule 3). □

6.4.4 Handling large state spaces

IOCO checking of two conforming labeled transition systems requires the comparison of their whole state spaces. For large industrial specifications this is often infeasible.

In the following we propose two techniques to overcome this problem. The first approach considers conformance within a fixed bound, i.e. we limit the depth of the conformance check. The second approach exploits the knowledge where the fault has been inserted in the LOTOS specification. By marking the place of the mutation it is possible to focus the conformance check on the relevant part of the state space only, i.e. the part that reflects the mutation. This approach still considers conformance within a particular bound. However, the limit of the conformance check is dynamically adopted such that the length of the preamble leading to the mutation does not matter. As we use a special test purpose to construct the specifications relevant part, we call this second approach *slicing-via-test-purpose*.

Bounded Input-Output Conformance

As we can construct the state spaces of the specification and its mutant on-the-fly, one possibility for handling large state spaces is to introduce a bound for the conformance check. Consequently, we apply the conformance check not to the whole specification's state space but only up to a particular depth. Thus, similar to bounded model checking (Biere et al., 2003), we check for bounded input-output conformance:

Definition 6.10 (Bounded input-output conformance) *Given a set of inputs L_I and a set of outputs L_U then* $\mathbf{ioco}^{|k|} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ *is defined as:*

$$IUT\ \mathbf{ioco}^{|k|}\ S =_{df} \forall \sigma \in Straces(S) \bullet (length(\sigma) \leq k) \Rightarrow (out(IUT\ \mathbf{after}\ \sigma) \subseteq out(S\ \mathbf{after}\ \sigma))$$

Example 6.5. For example, let the input-output transition system k of Figure 3.3 be the specification and let the IOTS l of Figure 3.3 be the model of an implementation. l does not (**ioco**-) conform to k, i.e. $\neg(l\ \mathbf{ioco}\ k)$, because $out(l\ \mathbf{after}\ \langle ?1, \delta, ?1\rangle) = \{!c, !t\} \not\subseteq \{!t\} = out(k\ \mathbf{after}\ \langle ?1, \delta, ?1\rangle)$. However, we have $l\ \mathbf{ioco}^{|0|}\ k$, because $out(l\ \mathbf{after}\ \langle\rangle) = \{\delta\} = out(k\ \mathbf{after}\ \langle\rangle)$. Furthermore, we have $l\ \mathbf{ioco}^{|1|}\ k$, because $l\ \mathbf{ioco}^{|0|}\ k$, $out(l\ \mathbf{after}\ \langle\delta\rangle) = \{\delta\} = out(k\ \mathbf{after}\ \langle\delta\rangle)$, $out(l\ \mathbf{after}\ \langle ?1\rangle) = \{!c, \delta\} = out(k\ \mathbf{after}\ \langle ?1\rangle)$, and $out(l\ \mathbf{after}\ \langle ?2\rangle) = \{\delta\} = out(k\ \mathbf{after}\ \langle ?2\rangle)$. We also have $l\ \mathbf{ioco}^{|2|}\ k$, because $l\ \mathbf{ioco}^{|0|}\ k$ and $l\ \mathbf{ioco}^{|1|}\ k$ and there is no trace of length two after which an output of l is not allowed by k. The shortest trace leading to non-conformance has a length of three, i.e. $\langle ?1, \delta, ?1\rangle$. Thus, with a bound greater or equal to three l does not conform to k, i.e. $\neg(l\ \mathbf{ioco}^{|3|}\ k)$. □

Bounded input-output conformance checking applies to any mutation operator. If we find a counter-example when checking for $S^m\ \mathbf{ioco}^{|k|}\ S$ within a particular bound k, then the counter-example is also valid for showing non-conformance for **ioco**. However, failing to show non-conformance within a particular bound k does not mean that we necessarily have an ioco-correct, i.e. an equivalent, mutant. Thus, the technique is sound but incomplete. Here, soundness guarantees that no counter-examples for equivalent mutants are generated. Hence, we will never produce redundant test cases from equivalent mutants. Due to incompleteness we may miss some test cases aiming for faults that are observable only above the boundary.

Slicing-via-Test-Purpose

A test case for fault-based testing consists of a preamble that aims to bring the implementation to a certain state in which the difference between the mutant and the original specification can be observed. When using

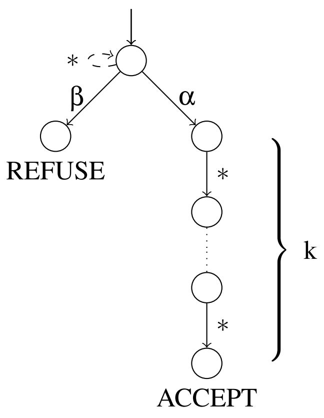


Figure 6.9: Test purpose for extracting the relevant part of specification with respect to a certain mutation.

a bounded input-output conformance check, the whole state space (up to a certain depth) is considered. This can be improved by considering only paths of the specification that lead to the mutation.

The idea is to extract a slice from the specification that includes the relevant preambles and the possibly faulty parts only. The relevant preambles for our conformance check are the traces leading to the places where the fault has been introduced in the mutant. Fortunately, we know where the specification has been mutated. Hence, the key idea is to mark the place of mutation in the LOTOS specification *S* with additional labels (α, β) that are not used in the initial alphabet of *S*.

For calculating a preamble, we insert an α event in front of the mutated LOTOS event, i.e. before the event that is changed by the mutation operator. Furthermore, we insert a unique β event after every other event of the LOTOS specification. Then, the slices can be calculated using TGV and the test purpose illustrated in Figure 6.9. This test purpose only selects (accepts) traces comprising α-labeled transitions and refuses β-labeled ones. Furthermore, it selects *k* actions after the preamble. These actions possibly exhibit the injected fault. The result of applying this *slicing-via-test-purpose* technique are two test processes (graphs), one for the original specification, and one for the mutant. Finally, the discriminating sequence is extracted from the two test processes that reflect the relevant behavior of their models.

Note that we still check for bounded conformance. As in general we cannot syntactically decide on the effect of the mutation we can only insert a mark that enables the calculation of the preamble. After the preamble we use a bounded check for an observable difference between the mutant and the specification. More formally, let $length(p_m)$ be the length of a mutation's preamble, then by the use of the slicing-via-test-purpose approach we check for $\mathbf{ioco}^{|(length(p_m)+k)|}$. *k* denotes the upper bound on the length of the postamble, i.e, mutants exhibiting their faulty behavior after the preamble within *k* actions will be detected.

Example 6.6. Figure 6.10 illustrates the application of a mutation operator that swaps two events on a LOTOS specification. The order of the two events *g2* and *g1* at Line 3 has been changed from *g2; g1;* (original) to *g1; g2;* (mutant). Both versions of the specification have been annotated with α and β. For the sake of simplicity this example does not show all the added β events. The underlying labeled transition systems of the specification and the mutant are depicted in Figure 6.11.

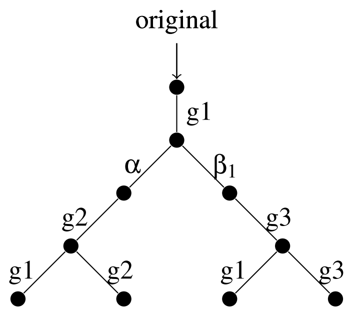
By the use of the test purpose illustrated in Figure 6.9 (with k=2) we extract the two test graphs from these labeled transition systems. Figure 6.12 illustrates the extracted test graphs. Recall that test graphs are deterministic, i.e. the mutated test graph has only one *g2* event leading to a pass state.

Finally, extracting the distinguishing sequence between the two test graphs leads to a new test purpose. This test purpose can then be used to generate a test case that will fail on an implementation that implements the mutant. □

```
process original[g1,g2,g3,α,β] : exit
  g1; (
    α; g2; ( g1; exit [] g2; exit )
    []
    β1; g3; ( g1; exit [] g3; exit )
  )
endproc
```

```
process mutant[g1,g2,g3,α,β] : exit
  g1; (
    α; g1; ( g2; exit [] g2; exit )
    []
    β1; g3; ( g1; exit [] g3; exit )
  )
endproc
```

Figure 6.10: Applying an event swap mutation operator to a LOTOS specification.



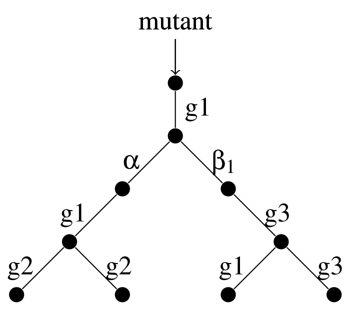
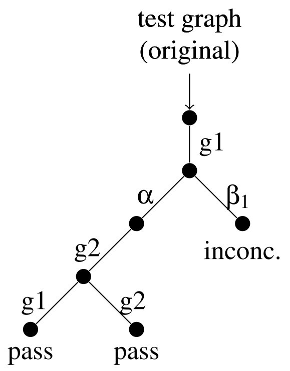


Figure 6.11: LTS representation of the original and the mutated specification.



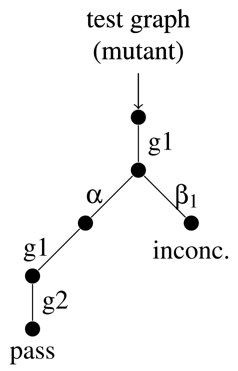
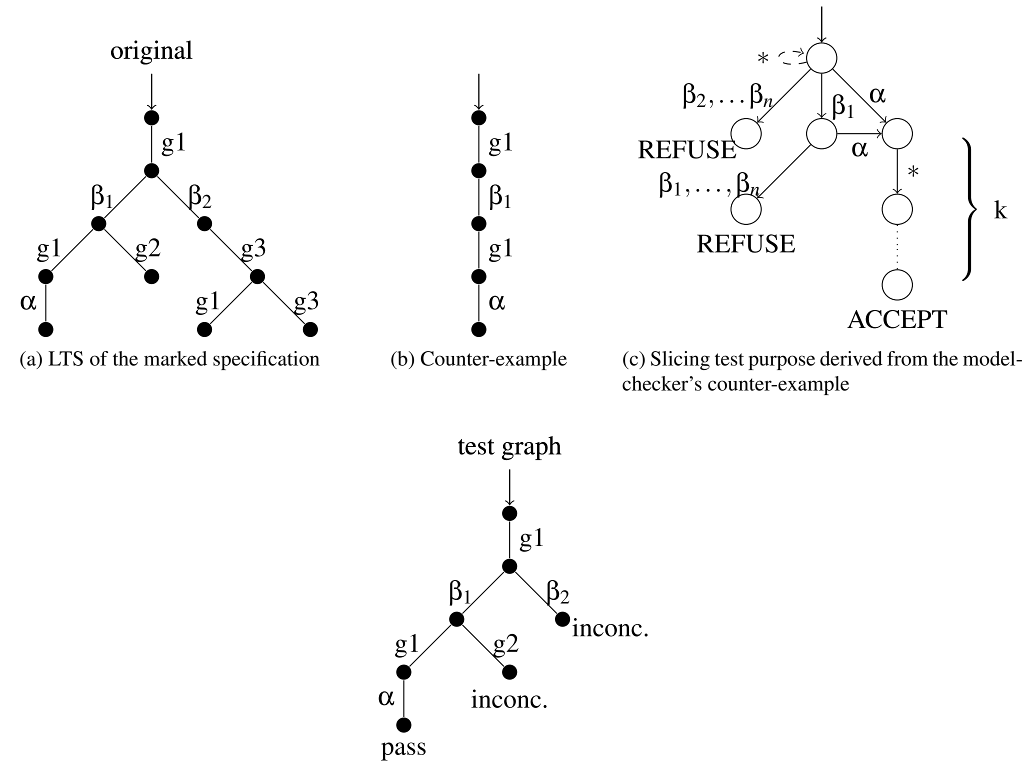


Figure 6.12: Obtained test graphs when applying the test purpose of Figure 6.9 (with k=2) to the original and the mutated LTS of Figure 6.11.

However, because of our insertion strategy of β and α there might be no β-free path leading to α. In that case the application of the test purpose of Figure 6.9 leads to an empty test graph. This basically means that our slicing strategy (our test purpose) was too strong. Thus, we have to allow more β-actions before α in the *slicing-test-purpose*.

Basically we need a path ending in α and containing all βs on the path to α. This path can be calculated using an on-the-fly model-checker and a safety property, i.e. a property that informally states that "something bad never happens". If we state that α is not reachable, a model-checker's counter-example is a trace that ends in α. Since we previously inserted α such a trace always exists if the specification contains no unreachable parts. The counter-example returned by the model-checker is a trace that contains all βs which should be allowed before α. If we make the β events distinguishable, e.g. by adding an unique number to each β event, the counter-example reflects the β events that should be allowed by the *slicing test purpose*.

Example 6.7. For example, running the model-checker on the labeled transition system depicted in



(a) LTS of the marked specification (b) Counter-example (c) Slicing test purpose derived from the model-checker's counter-example

(d) Resulting test graph

Figure 6.13: LTSs illustrating the *slicing-via-test-purpose* technique.

Fig. 6.13a results in the counter-example shown in Fig. 6.13b. This counter-example says that we should allow the event β_1 in order to reach α. Translating this counter-example results in the test purpose illustrated in Fig. 6.13c. If we apply this test purpose (with $k = 0$) to the labeled transition system of Fig. 6.13a, the result looks like the LTS depicted in Fig. 6.13d. □

The model-checker returns a single trace ending with an α transition. This is sufficient for mutations that show their effect after every preamble ending in α, which is the case if a mutation operator directly affects a LOTOS action (i.e. the action mutation operators of Table 6.1: CRO, EDO, EIO, ERO, ESO, HDO, and USO). For these operators any preamble will lead to the failure exhibiting part of the specification's underlying LTS. However, if data is involved in the mutation, e.g. if guards are mutated, then not every preamble necessarily leads to the faulty part of the specification's LTS. For example, consider a LOTOS action like `out !var` where *var* is a Boolean variable. Applying the Operand Replacement Operator (ORO) may lead to `out !false`. This mutant only differs from the original in the cases where *var* is *true*.

In other words, if we are unlucky the model-checker may return a trace leading to α where the mutant does not exhibit its faulty behavior. While our α insertion strategy ensures that this cannot happen for action mutations, this problem occurs for data mutations (i.e. ASO, ENO, LRO, MCO, ORO, RRO, and SNO) and for structural mutations (i.e. POR, PRO, SOR).

For data mutations we can force the model-checker to return a counter-example that leads to the faulty part of a mutant. The idea is to attach the original and the mutated expression to the α mark and state that these two expressions are unequal. We then run the model-checker with the safety property claiming that $\alpha!true$ is not reachable.

Example 6.8. Consider the example process illustrated in Figure 6.14. This process reads a natural

```
1 process original[input,output,α] : exit
2   input ?number:Nat;
3   α !((number < 5) ne (0 < 5));
4   [number < 5] -> (
5     output !5;
6   )
7 endproc
```

```
1 process mutant[input,output,α] : exit
2   input ?number:Nat;
3   α !((number < 5) ne (0 < 5));
4   [0 < 5] -> (
5     output !5;
6   )
7 endproc
```

Figure 6.14: Applying the operand replacement operator (ORO) to a guard of a LOTOS specification.

number and outputs five if the read number is lower than five. The operand replacement operator (ORO) changes the guard in Line 4 from `number < 5` to `0 < 5`. By adding `((number < 5) ne (0 < 5))` to the α action (see Line 3) the specification's labeled transition systems comprises $\alpha!true$ and $\alpha!false$ events. By using a preamble that leads to $\alpha!true$ we can ensure that the difference between the mutant and original specification is exhibited after the preamble. □

However, if the data mutation results in an equivalent mutant, i.e there is no $\alpha!true$ labeled transition, then the model-checker needs to traverse the whole specification's state space. Furthermore, this technique cannot be applied to structural mutations. For the structural mutations we would need more information about which data values would reveal a mutant's faulty behavior (if any).

Remark 6.1 *In the case of non-determinism, our slicing-via-test-purpose approach may fail to identify equivalent mutants. A specification may have a non-deterministic branch that has exactly the behavior of the introduced mutation. That is, the behavior of the mutant is allowed by the specification. However, if this behavior is sliced away by our test purpose, then our approach wrongly detects a non-equivalent mutant.*

Remark 6.2 *The use of $*$-labeled transitions within the test purpose for the extraction of the postamble imposes some limitations on our slicing-via-test-purpose approach. If the test purpose selects a postamble of length k, then there has to be trace starting after the inserted α of a length of least k events. If there is no such postamble then* TGV *fails to extract a test case. Thus we cannot apply mutations that for example insert* `stop` *events, because there are no transitions (except δ-labeled ones) after a* `stop` *statement in the underlying LTS of a* LOTOS *specification. This limitation is because we rely on* TGV*; as a design decision this tool does not select δ-labeled transitions using a $*$-labeled transition within a test purpose.*

6.4.5 Overall Approach

Using the findings of Section 6.4.2 and the techniques for handling large state spaces of Section 6.4.4 we can now reformulate the test purpose generation procedure of Aichernig and Corrales Delgado (2006). Thus, we generate a test purpose for a specification $S = (Q, L, \rightarrow, q_0)$ as follows:

1. Select a mutation operator O_m.
2. Generate a mutated version S^m of the specification S by applying O_m.
3. Extract a discriminating sequence c by
   - using bounded input-output conformance checking.
   - using slicing-via-test-purpose, which is implemented by
     - Using the knowledge where O_m changes the specification to generate S' by inserting markers $\{\alpha, \beta\} \not\subseteq L$ into the formal specification S.
     - Call the model-checker to generate a trace that ends in α (using CADP-evaluator (Mateescu and Sighireanu, 2000)) and derive a *slicing-test-purpose* from this trace.
     - Generate two complete test graphs, CTG_α for the specification and CTG_α^m for the mutant, by the use of the *slicing-test-purpose* (using TGV).

   - Transform CTG_α and CTG_α^m to CTG and CTG^m by hiding the marker labels α and β (using CADP-Bcg (Garavel et al., 2002)).
   - Check CTG and CTG^m for input-output conformance (using an ioco checker).

4. The counter-example c, if any, gives the new test purpose*.
5. This test purpose leads to a test case that fails on an implementation that implements the mutant.

6.5 Coverage-based Test Purposes

Besides using mutations for the generation of test purposes, another well-known test case selection technique is the use of coverage criteria. While Section 6.3.5 uses coverage on complete test graphs, another way is to apply coverage criteria on the level of the specification. In general, a coverage criterion defines a set of test requirements that should be exercised. The coverage value expresses the percentage of the test requirements that are actually covered by a given test suite.

In this section we define coverage criteria for LOTOS specifications, and show how the LTS representing the semantics of a LOTOS specification has to be extended in order to allow coverage of the defined criteria. Finally, we present a generic set of test purposes which allow one to generate test cases achieving maximum specification coverage.

6.5.1 Basic Coverage of LOTOS Specifications

The behavioral part of a LOTOS specification is expressed in terms of processes and their communication. A process is considered to be a black-box at some level of abstraction where only the process's external behavior is considered. Similar to function coverage for sequential programs, process coverage gives the percentage of processes executed within a LOTOS specification.

Definition 6.11 (Process Coverage) *A process of a* LOTOS *specification is* covered*, if it is executed. The process coverage represents the percentage of processes of the specification that are covered.*

The behavior of a LOTOS process consists of actions. The idea of action coverage for LOTOS specifications is that every action statement in the specification should be executed at least once, similar to statement coverage (Myers, 1979) for source code.

Definition 6.12 (Action Coverage) *An action of a* LOTOS *specification is* covered*, if it is executed. The action coverage represents the percentage of actions of the specification that are covered.*

A single action statement within the specification may generate several edges within the underlying LTS, because data is handled by enumeration. If an action has parameters, e.g., a Boolean variable, then the LTS contains all possible enumeration of the parameters' values. For one Boolean parameter the LTS contains two edges: one labeled with true and another one labeled with false. Action coverage only requires that one of these edges is taken by a test case.

Example 6.9. The specification illustrated in Figure 3.11 has actions in Lines 3, 7, 19, 24, 29, and 32. The test case shown in Figure 6.15 covers 66.6% of these actions: The action in Line 24 is covered by the first two transitions `ui!1` and `ui!2`. The transition `ui!display` covers the action of Line 7. The final two transitions `out!2` and `out!1` cover the two actions of Line 32 and of Line 29, respectively. The process coverage of this test case is 100.0% because every process (`Main` and `DisplayStack`) has been entered at least once. Note that in this example the edges leading to fail states do not add to action nor to process coverage, because there is no test run on our specification that leads to a fail verdict state. □

*The labels of the test processes are marked with *INPUT* or *OUTPUT*. We remove these marks. Furthermore, we have to add Refuse and Accept states.

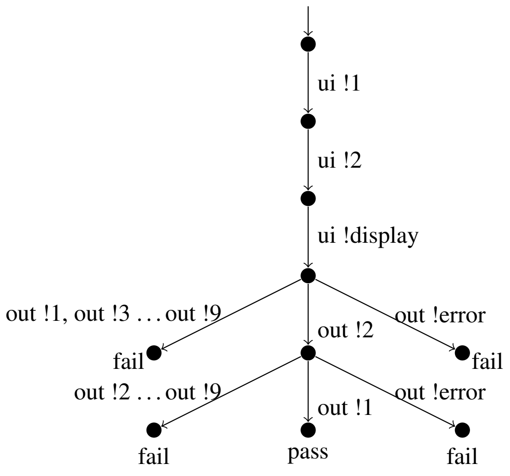


Figure 6.15: Test case having 100.0% process coverage and 66.6% action coverage on the specification of Figure 3.11.

6.5.2 Logical Coverage Criteria for LOTOS Specifications

LOTOS specifications contain Boolean expressions within guards. For example, Line 9 in Figure 3.11 contains the logical expression `(op eq display) and (size(s) gt 0)`. Following a large body of literature of coverage in software testing, we call such an expression a *decision*. A decision comprises a number of *conditions* (also known as *clauses*), that are connected by logical operators. In this section we adapt several known coverage criteria for logical expressions to LOTOS specifications.

Definition 6.13 (Decision Coverage) *A decision is covered if it evaluates to true and to false at some point during test execution. The decision coverage represents the percentage of decisions in a* LOTOS *specification that are covered.*

Decision coverage has also been called *branch coverage* (Myers, 1979) or *predicate coverage* (Ammann et al., 2003). In contrast to action coverage, each decision results in two test requirements, one for the decision to evaluate to true and one for it to evaluate to false; this applies to the condition and to the condition/decision criterion as well.

Example 6.10. The specification given in Figure 3.11 has five decisions on Lines 9, 13, 17, 28, and 31. The test case of Figure 6.15 achieves 70% decision coverage with regard to these decisions: The transition `ui!display` and the fact that the stack size is changed by the two predecessor transitions cause the guard of Line 9 to evaluate to true and the guards of Line 13 and of Line 17 to evaluate to false. The last two transitions `out!2` and `out!1` cause the decisions of Line 28 and Line 31 to evaluate to true and to false, respectively. The transitions `ui!1` and `ui!2` do not directly trigger any guard. □

A decision consists of conditions separated by logical operators (e.g. and, or); these conditions are considered by condition coverage (also known as clause coverage (Ammann et al., 2003)).

Definition 6.14 (Condition Coverage) *A decision consists of conditions separated by logical operators (e.g. and, or). A single condition is covered if it evaluates to both true and false at some point during test execution. The condition coverage represents the percentage of conditions in a* LOTOS *specification that are covered.*

Example 6.11. The specification listed in Figure 3.11 has ten conditions on Lines 9, 13, 17, 28, and 31. The transition `ui!display` of the test case illustrated in Figure 6.15 lets the first condition of Line 9

Table 6.2: Modified condition/decision coverage for the guard `((op eq add) and (size(s) lt Succ(Succ(0)))) or ((op eq display) and (size(s) eq 0))` of Figure 3.11.

TC	C_1	C_2	C_3	C_4	$(C_1 \wedge C_2) \vee (C_3 \wedge C_4)$
	`op eq add`	`size(s) lt Succ(Succ(0))`	`op eq display`	`size(s) eq 0`	
1	T	F	T	F	F
2	F	T	T	F	F
3	T	T	T	F	T
4	T	F	F	T	F
5	T	F	T	T	T

evaluate to true. Because the two transitions `ui!1` and `ui!2` raise the stack size to two, the transition `ui!display` also makes the condition `(size(s) gt 0)` and `(size(s) ge Succ(Succ(0)))` true. Furthermore, the conditions `(op eq add)`, `(size(s) lt Succ(Succ(0)))`, and `(size(s) eq 0)` evaluate to false. The two transitions `out!2` and `out!1` leading to the pass state make the conditions of Lines 28 and 31 evaluate to false and true, and to true and false, respectively. Thus, the condition coverage is 60%□

As satisfying condition coverage does not guarantee that all decisions are covered (i.e., decision coverage is not subsumed by condition coverage), it is common to define the condition/decision (CD) coverage criterion as the combination of decision and condition coverage.

Definition 6.15 (Condition/Decision Coverage) *The condition/decision coverage represents the percentage of conditions and decisions in a* LOTOS *specification that are covered.*

This means that 100% condition/decision coverage is achieved if all conditions evaluate to true and to false, and if every decision also evaluates to true and to false during the test execution.

Example 6.12. Our stack calculator specification includes ten conditions and five decisions (Lines 9, 13, 17, 28, 31). Counting the condition/decision coverage of our test case shown in Figure 6.15, we get a CD coverage of 63%. □

Another coverage criterion for logical expressions is the modified condition/decision coverage (MCDC) (Chilenski and Miller, 1994). The essence of modified condition/decision coverage is that each condition must show that it independently affects the outcome of the decision. That is, test cases must demonstrate that the truth value of the decision has changed because of a change of a single condition.

Definition 6.16 (Modified Condition/Decision Coverage) *The modified condition/decision coverage represents the percentage of conditions that independently effected the outcome of a decisions in a* LOTOS *specification during execution.*

For each condition a pair of test cases is needed, i.e., one making the decision true while the other makes the decision false. However, by overlapping these pairs of test cases usually $N+1$ test cases are sufficient for covering a decision comprising N conditions.

Example 6.13. Table 6.2 depicts a minimal set of truth values for the conditions of the guard of Line 17 in Figure 3.11. A set of test cases that cause the conditions to take these truth values has 100% MCDC coverage on the guard of Line 17. For example, consider the two test cases 1 and 5. They only differ in the truth value of the condition C_4. However, the truth value of the decision is different for the two test cases 1 and 5 as well. Thus, this test case pair covers C_4 with respect to MCDC coverage. Furthermore, C_3 is covered by the test cases 4 and 5, C_2 is covered by the test cases 1 and 3 and C_1 is covered by the two test cases 2 and 3. □

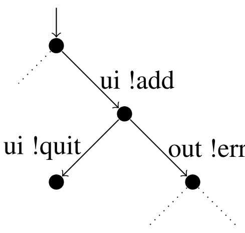


Figure 6.16: Part of the underlying LTS of the stack calculator specification.

6.5.3 Weak and Strong Coverage

Due to the inherent parallel nature of LOTOS specifications we need to distinguish between weak and strong coverage. At any point of execution one may choose between different actions offered by a LOTOS specifications. Obviously, the offered actions depend on the structure of the specification.

For example, consider a specification having two processes $P1$ and $P2$ in parallel, i.e., $P1|||P2$. The $|||$ parallel execution operator denotes any interleaving of the actions offered by $P1$ and $P2$. Thus, at the very beginning of this specification fragment one can choose between the actions initially offered by $P1$ and the actions initially offered by $P2$. A similar situation can be constructed for guarded expressions. An action may be offered only if a particular guard evaluates to true.

A coverage item is covered weakly if the actions related to that coverage item are offered by the specification at some moment during the execution of a test case. Weak coverage does not require that a test case synchronizes on the relevant actions. More formally, weak coverage is defined as follows:

Definition 6.17 (weak coverage) *Given a set of actions $C = \{a_1, \dots, a_n\}$ comprising the actions relevant for a particular coverage item, an LTS representing a test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, and a specification $S = (Q^S, L^S, \rightarrow_S, q_0^S)$, then t weakly covers S with respect to g, iff*

$$\exists a \in C \bullet \exists \sigma \in traces(t) \bullet a \in init(S \textbf{ after } \sigma)$$

Depending on the coverage criterion the relevant actions differ. In the case of process coverage the relevant actions C are given by the transition labels of the transitions that are enabled in the LTS because of the process's behavior. For action coverage the relevant actions C are given by the transition labels that result from the action to be covered. For logical coverage criteria we have to distinguish between the $true$ and the $false$ case. In one of these cases there are more transitions enabled in the LTS than in the other. Without loss of generality assume that the $true$ case has more enabled transitions. Then, C for the $true$ case is given by the transition labels that are absent in the $false$ case. The actions C for the $false$ case are all actions that are enabled in the $false$ case.

We conclude that we have strong coverage if the actions of interest are not only offered by the specification, but if the test case synchronizes on one of those actions. That is, if one action is taken by the test case. More precisely, strong coverage for a coverage goal, expressed in terms of actions, is defined as follows:

Definition 6.18 (strong coverage) *Given a set of actions $C = \{a_1, \dots, a_n\}$ comprising the actions relevant for a particular coverage item, an LTS representing a test case $t = (Q^t, L^t, \rightarrow_t, q_0^t)$, and a specification $S = (Q^S, L^S, \rightarrow_S, q_0^S)$, then t strongly covers S with respect to g, iff*

$$\exists \sigma \in traces(t) \bullet C \cap init(t||S \textbf{ after } \sigma) \neq \emptyset$$

As this definition shows, strong coverage only requires that at least one of the relevant actions is taken by the test case.

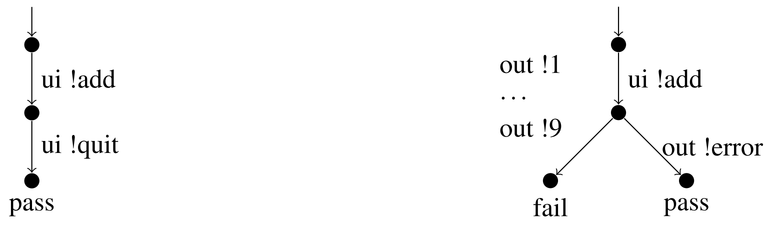


Figure 6.17: Two different test cases illustrating weak and strong coverage.

Table 6.3: Subsumption relation of the approached coverage criteria.

	Process		Action		Decision		Condition		CD		MCDC	
	(W)	(S)	(W)	(S)	(W)	(S)	(W)	(S)	(W)	(S)	(W)	(S)
Process (W)		$\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Process (S)	$\not\subseteq$		$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Action (W)	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Action (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Decision (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\not\subseteq$	$\not\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$
Decision (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\subseteq$	$\not\subseteq$	$\subseteq$
Condition (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$
Condition (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\subseteq$	$\not\subseteq$	$\subseteq$
CD (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\subseteq$	$\subseteq$
CD (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\subseteq$
MCDC (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$
MCDC (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	

Example 6.14. This example serves to illustrate the difference between weak and strong coverage. Figure 6.16 depicts the initial part of the underlying labeled transition system of our stack calculator specification (Figure 3.11). As this LTS shows, after executing `ui!add` (Line 7) the calculator may either issue an error (`out!error`) or the user may quit (`ui!quit`) the calculator.

For this part of the specification two different valid test cases are possible; these test cases are illustrated in Figure 6.17. Both test cases enable the block guarded by the condition in Line 17. However, the test case on the left hand side does not take an action from the guarded block, but chooses the quit action. This test case covers the decision's true case of the guard weakly. On the other hand, the test case illustrated on the right of Figure 6.17 chooses an action that is possible only because the guard has evaluated to true. Thus, this test case achieves strong coverage with respect to the conditions and decisions of interest. □

6.5.4 Relation between Coverage Criteria

Table 6.3 relates the coverage criteria of Section 6.5.1 and of Section 6.5.2 with respect to weak (W) and strong (S) coverage. The table is read from left to top. For example, the intersection of the first row with the second column states that weak process coverage (Process (W)) is subsumed by strong process coverage (Process (S)).

As this table shows, neither weak nor strong process coverage subsume any other coverage criteria. This is also true for weak and strong action coverage. In particular, action coverage does not subsume process coverage because LOTOS allows one to use processes which do not comprise actions.

Furthermore action coverage neither subsumes any other coverage criterion nor is subsumed by any other coverage criterion. Let *a*, *b*, and *c* be actions and let `a [] b [] [x>5]`$\rightarrow$`(c)` be the specification. Then a test suite having 100% action coverage consists at least of three test cases while the test suites for our logical criteria comprise only two test cases.

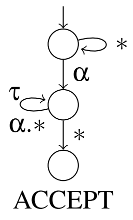


Figure 6.18: Test purpose for generating test cases with respect to process coverage of marked specifications.

Weak coverage of any logical criteria is always subsumed by its strong pedant. On the contrary, weak coverage never subsumes strong coverage. This follows from the definitions of weak and strong coverage. Thus, weak condition decision coverage does not subsume strong decision coverage. However, weak CD coverage subsumes weak decision coverage and strong CD coverage subsumes strong decision coverage.

Strong MCDC subsumes all of our coverage criteria except process and action coverage. This is because for each condition MCDC comprises one test case such that condition evaluations to both true and false. In addition, there is at least one test case making the decision true and at least one test case making the decision false.

6.5.5 Deriving Test Purposes From Coverage Criteria

We have defined the coverage criteria on the syntactical level of the specification while test purposes operate on the semantic level, e.g., on the labeled transition system. Unfortunately, the LTS does not explicitly contain the information needed to generate coverage based test purposes. In order to bridge this gap we have to annotate the LOTOS specification with additional information such that this information is also visible in the LTS.

More specifically, we insert probes α that are not in the initial alphabet of the specification S, i.e., $\alpha \notin L^S \cup \{\tau\}$; we call a specification with inserted probes *marked*. Each probe results in a distinct transition in the labeled transition system, and therefore makes coverage relevant information of the specification visible. By selecting α labeled transition in the test purpose we can now derive test purposes with respect to a certain coverage criterion. Note that for the final test case the α label has to be hidden again.

In the following we explain for every coverage criterion what probes are needed and what the test purposes look like.

Process Coverage For process coverage we generate as many marked specification as there are processes within the specification. Each marked specification comprises a single probe which is inserted directly after the process declaration. In the case of our stack calculator example we generate 2 different copies. In one copy there is an α-probe at Line 6, while in the other copy the α-probe is inserted at Line 28 (in front of the guard).

By using the test purpose depicted in Figure 6.18 we derive test cases for each of the specifications in order to obtain a test suite for process coverage. This test purpose simply states that any sequence covering an α-labeled transition followed by an arbitrary action is a valid test case. (Alternatively, one could use a single specification with several probes without any loss of generality. However, our experiments have shown that this slows down TGV significantly.)

Action Coverage In order to generate test cases with respect to action coverage we construct as many marked specifications as there are actions within the specification. For example, for our stack calculator specification we generate 5 different copies. In each copy we insert an α after the action of interest, e.g.,

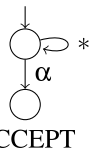


Figure 6.19: Test purpose for generating test cases with respect to action coverage of marked specifications.

```
process DisplayStack[out,α](s:Stack):exit:=
 α !(size(s) eq Succ(0));
 (
   [size(s) eq Succ(0)] -> (out !top(s); exit)
   []
   [size(s) gt Succ(0)] ->
     (out !top(s); DisplayStack[out](pop(s)))
 )
endproc
```

Figure 6.20: Stack calculator specification with probe to cover a condition.

after the action statements on Lines 7, 19, 24, 29, and 32. Then, the test purpose depicted in Figure 6.19 is used with each of the specifications in order to derive test cases with respect to action coverage. This test purpose simply states that any sequence covering a transition labeled with α is a valid test case; because α follows after the considered action. This guarantees that the action was executed.

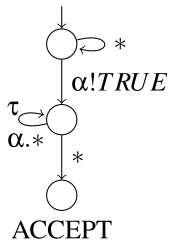
Decision Coverage For generating test cases with respect to logical coverage criteria we have to equip our α probes with additional information, which allows us to select a certain outcome of a decision. In the case of decision coverage we simply add the whole decision to the probe.

Figure 6.20 serves to illustrate this technique. This figure shows the DisplayStack-process of our stack calculator specification. In order to generate test cases for covering the condition of Line 4, we added a probe equipped with the decision in Line 2. The underlying LTS of this piece of specification contains the edges α`!TRUE` and α`!FALSE`, which can be selected directly by the test purpose.

Note that we insert the probes before the decisions if they are not part of a choice operator (i.e., "[]"). In the case of choices we insert the marker before the first condition of the choice (e.g., the probe for the condition of Line 6 in Figure 6.20 is inserted at Line 2). For each marked specification we get two test cases from the two test purposes of Figure 6.21, one for each possible outcome of a decision. In order to ensure that the decision is evaluated we need to select one non-internal edge, e.g., an edge not labeled with τ. Therefore, we have an '∗'-labeled edge leading to the accept state, which selects an edge which is possible after α.

Condition Coverage In order to generate test cases with respect to condition coverage we insert our α probes in the same manner as for decision coverage. Instead of equipping the probes with the whole decision we split the decision into its conditions. For example, the probe for covering the two conditions in the guard in Line 9 of our stack calculator example is α `!(op eq display) !(size(s) gt 0)`. The corresponding test purposes are illustrated in Figure 6.22. For each condition of our inserted markers (α) we need to select the true (!TRUE) and the false (!FALSE) outcome while we do not care about the other conditions (![A-Z]∗). Since we need to select a particular condition we have to add ![A-Z]∗ at the right position for every other condition in the label of the test purpose.

If there are n conditions in a decision we need $2 \times n$ test purposes in order to derive test cases with respect to decision coverage. As for decision coverage we need to select one edge after the α probe in order to ensure that the condition is evaluated.



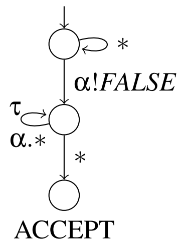
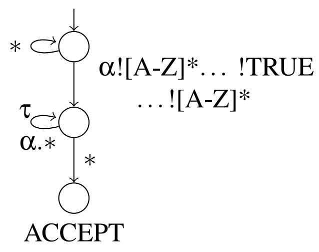


Figure 6.21: Test purposes for generating test cases with respect to decision coverage of marked specifications.



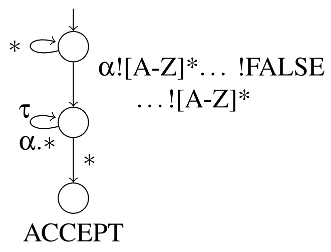


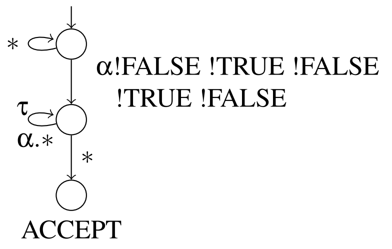
Figure 6.22: Test purposes for generating test cases with respect to condition coverage of marked specifications.

Condition/Decision Coverage We generate test cases with respect to CD coverage by inserting probes similar to condition coverage. The probes are equipped with the decision as the first element followed by the conditions of the decision. We then use the test purposes of Figure 6.22 and instantiate them for all parameters of the probe, i.e., for the decision and for all conditions.

Modified Condition/Decision Coverage The probes inserted into specifications for deriving MCDC based test purposes are equivalent to the probes inserted for condition/decision coverage. On the contrary, the test purposes do not contain "![A-Z]"-elements which match any logical value, e.g., true and false, but each label of a test purpose matches a particular Boolean value of every condition. There are several possibilities to calculate truth values for conditions that result in MCDC pairs; we refer to Ammann and Offutt (2008) for an overview. In order to generate a minimal number of test purposes for a decision we calculate the truth-table for that decision. Then we search for all sets of truth-value assignments to the conditions of the decision such that each set is a valid MCDC test suite of minimal length. From these sets we randomly select one set which then forms our test suite. The truth-value assignments are directly used within the test purposes. In total, we generate $N+1$ different test purposes for N conditions. Each test purpose selects a particular valuation of the conditions and of the decision of a probe. For example, let Table 6.2 be the selected set of truth-value assignments for the decision `((op eq add) and (size(s) lt Succ(Succ(0)))) or ((op eq display) and (size(s) eq 0))` in Line 17 of Figure 3.11. Then the generated test purposes for C_1 and C_2 of Table 6.2 look like those illustrated in Figure 6.23. We omit the test purposes for C_3 and C_4 here.

Remark 6.3 *Except for action coverage, our test purposes can guarantee weak coverage only. A solution for achieving strong coverage would be to insert other probes after the actions relevant for a particular coverage item. For example, for process coverage one could insert a β after every action that can occur as the first action within a process.*

However, while we can always decide where to insert regular α probes based on the syntax of the specification, this cannot be done on the syntactical level for β probes. We may need to evaluate parts of the specification before we can decide on the positions of β probes. However, if such probes are inserted



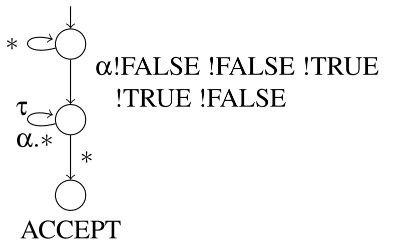


Figure 6.23: Test purposes for generating test cases with respect to modified condition/decision coverage of marked specifications.

into the specification, test purposes can make use of these β actions in order to ensure that the right branch within the LTS is taken.

For example, suppose we want to strongly cover a process that does not comprise any action but calls other processes. Further suppose, that these other processes do not return but call themselves recursively. In that case we need to insert β after the first action of the called processes.

6.5.6 Supporting manually designed test purposes

Our probe insertion technique does not only allow one to automatically generate test suites based on coverage criteria. It may also be used to complement manually designed test purposes.

Test cases generated for manually designed test purposes may not fully cover all aspects of a particular model. By the use of our probe-based technique we can determine the coverage value of a given set of test cases. Furthermore, we can automatically generate test cases that exercise parts of the specification missed by the test cases derived from manually designed test purposes.

More precisely, we do the following in order to complement test cases generated for manually designed test purposes. Given a test suite derived from a set of test purposes we run the test cases of that test suite on a specification comprising all probes for a particular coverage criterion. From these runs we can extract the set of covered probes. By generating test cases for the uncovered probes we can increase the overall coverage value of the test suite.

Example 6.15. Suppose we want to complement the manually designed test purpose illustrated in Figure 6.1 with respect to action coverage for our stack calculator specification. Assuming that the test case generated by TGV for this test purpose looks like the test case depicted in Figure 6.15, running it on a copy of the specification comprising all α probes for action coverage gives a coverage value of 66.6%.

From this test run we can deduce that the test case misses the probes corresponding to the action `ui!quit` in Line 3 and to the action `out!error` in Line 19. We can automatically generate two test purposes which aim to cover the two missed probes. These two test purposes will lead to test cases similar to the test cases illustrated in Figure 6.17. The complemented test suite comprises three test cases which in total achieve 100% action coverage on the specification. □

6.5.7 Reduction

In practice there is often insufficient time for thorough testing activities within industrial projects. Therefore it is reasonable to try to reduce the size of generated test suites. However, the effect of the reduction on the fault detection ability of the test suites should be small.

The coverage based techniques proposed in this section can be used to apply reduction during test case generation. A single test case may cover more than the coverage item it has been generated for. When using a probe based technique as described in this paper it is easy to identify all items covered by a particular test case. This is done by running the generated test case on a specification containing all probes.

An optimal test suite is a minimal set of test cases such that all items are covered. Unfortunately, this is equivalent to the set covering problem which is known to be NP-complete. We can, however, approximate the optimal test suite. After a test case has been generated we run this test case on a specification containing all probes and extract the covered probes. We skip test purposes for probes that are covered by previously generated test cases. Note that this minimizes both the number of test cases and the number of invocations of TGV. The number of generated test cases depends on the processing order of the different test purposes.

Note that there is still some potential for reducing the test suite sizes left. We currently do not consider that test cases may cover probes of previously generated test cases. This would possibly allow to remove some more test cases from the generated test suites.

Chapter 7

Analyzing Test Case Execution Results

The contents of this chapter have been published in the paper (Weiglhofer et al., 2009b).

Although the previous chapters have presented techniques that allow one to select a finite number of test cases, large models usually lead to large test suites. Because of the black-box view many of the generated test cases may fail because of the same failure within the software system. For example, when testing a Session Initiation Protocol Registrar (Aichernig et al., 2007), we experienced that the ratio between failed test cases and detected failures is very low, i.e., 0.18%: A set of 4944 failed test cases detected only 9 discrepancies between our specification and a commercial implementation.

In light of this the test result analysis becomes a tedious and time consuming task. Consequently it is our aim to assist the test engineer by grouping test cases with respect to the detected failures. In this chapter we investigate a technique that groups test cases such that test cases of one group most likely detect the same failure. This grouping supports test engineers and software developers in two ways:

- Test engineers do not need to analyze all failed test cases but only one per group. Presenting only the shortest test case within each group to the test engineer can further simplify the post analysis of testing.
- Software developers that need to fix the bugs do not need to rerun all test cases but only a small set of regression tests, i.e., only one test per group.

Our approach relies on spectrum-based fault localization (Zoeteweij et al., 2007) applied on the level of the formal models. Spectrum-based fault localization ranks elements of the specification according to the likelihood of being not correctly implemented. Then, failed test cases are grouped with respect to the suspected faulty parts, such that two test cases are in the same group if they execute the same potentially incorrectly implemented parts of the specification. Our experiments show that the groups generated by this approach resemble the detected failures nicely. For example, we were able to generate 43 groups for the 4944 test cases that detected 9 failures. With respect to our SIP Registrar case study, this means that only 43 test cases instead of 4944 test cases have to be analyzed to identify the 9 detected failures. This is an improvement of approximately 99%.

7.1 Spectrum-based Fault Localization

In this chapter we combine spectrum-based fault localization with model-based testing. Thus, this section briefly introduces spectrum-based fault localization.

```
specification StackCalc [ui,out]:exit
behavior
 Main[ui,out](nil) [> ui !quit ; exit
                         Block 1
where
 process Main[ui, out](s:Stack):noexit:=
 (
  ui ?  op :  Operator ;
                Block 2
  (
   [(op eq display) and (size(s) gt 0)]→(
     DisplayStack[out](s) >> Main[ui,out](s)
   )
   []
   [(op eq add) and
    (size(s) ge Succ(Succ(0)))]→(
     Main[ui,out](push(top(s)+top(pop(s)),
                      pop(pop(s))))
   )
   []
   [((op eq add) and (size(s) lt Succ(Succ(0))))
    or ((op eq display) and (size(s) eq 0))]→ (
      out !error ; Main[ui,out](s)
         Block 3
   )
  )
 )
 []
 ( ui ?  num:  Nat ; Main[ui,out](push(num,s)) )
            Block 4
 endproc

 process DisplayStack[out](s:Stack):exit:=
  [size(s) eq Succ(0)] →
     out !top(s) ; exit
        Block 5
  []
  [Size(s) gt Succ(0)] →
     out !top(s) ; DisplayStack[out](pop(s))
        Block 6
 endproc
endspec
```

Figure 7.1: LOTOS specification of a simple stack calculator.

The idea of spectrum-based fault localization is to identify the most likely spots in a program to contain faults given a set of passed and failed test runs. A program spectrum is an execution profile that indicates which parts of a program have been executed during a test run. Program spectra (Reps et al., 1997) have been used by several tools (e.g., Pinpoint (Chen et al., 2002), Tarantula (Jones et al., 2002), AMPLE (Dallmeier et al., 2005)) to localize faults. An overview of different forms of program spectra has been given by Harrold et al. (1998).

The data for a program spectrum is collected during run-time and typically comprises the execution counts for different parts of the program. There are various ways to partition a program and every method results in a different form of spectrum. We rely on various instantiations of the so-called block hit spectra (Abreu et al., 2007), also known as branch spectra (Harrold et al., 1998). A block hit spectra uses one boolean flag for every block of the program. Each boolean flag indicates whether the block has been executed or not. Thus, for each test case we get a so-called execution signature comprising the values of the hit flags of every block.

Example 7.1. Figure 7.1 shows one possibility of partitioning a LOTOS specification into blocks. Each gray highlighted part serves as a single block for the spectrum calculation of test runs. A more detailed discussion of how we select blocks for LOTOS specifications can be found in Section 7.3. □

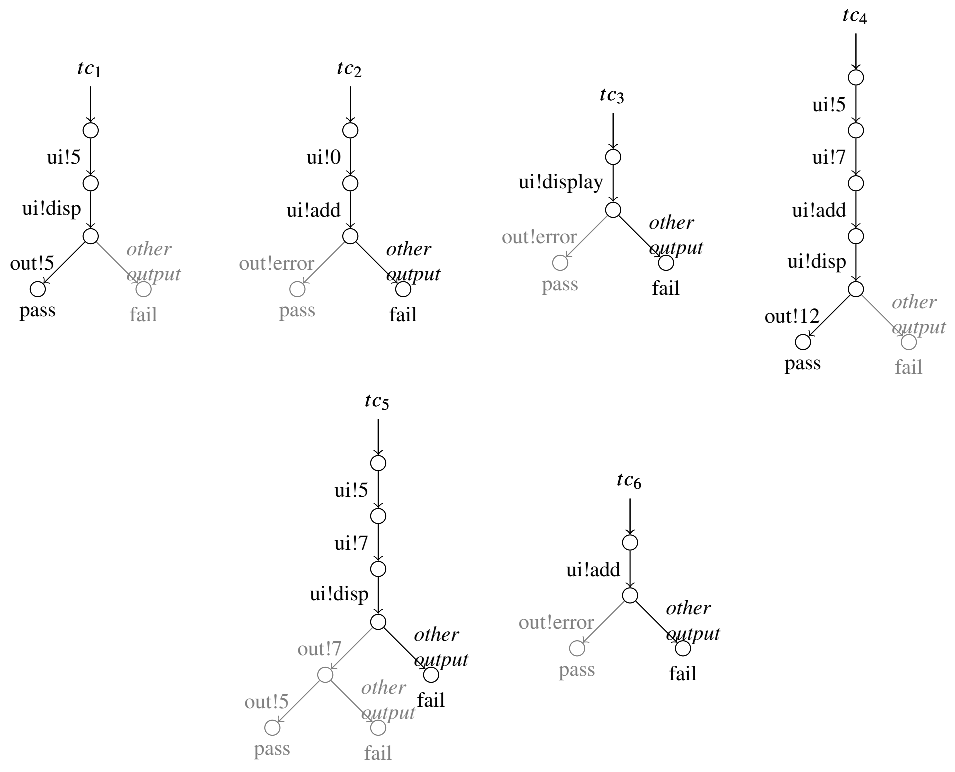


Figure 7.2: Six test cases (black and gray colored edges) and the resulting test runs (black colored edges) on an implementation for the stack calculator example.

Example 7.2. Figure 7.2 illustrates six different test cases that will be used throughout this chapter. The test runs on an implementation of the stack calculator are highlighted by the black colored edges. There are two detected failures within the implementation: First, our implementation does not raise an error when the stack does not meet the operator's requirements. This is detected by the test cases tc_2, tc_3, and tc_6. The second error is that when displaying the stack the implementation only prints the stack content if the stack size is equal to one. As the test case tc_5 tests for displaying a stack with two elements, it fails on this implementation. □

For fault localization with respect to the block hit spectra of test runs one calculates a binary matrix. As illustrated in Figure 7.3, this matrix comprises one column for each of the n identified blocks and one row for each of the m test runs. There is a '1' in a particular cell (i, j) if the corresponding block n_j has been hit by the test run m_i. Furthermore, the test result of the test runs are combined into an error vector. This vector comprises a '0' at row i if the test run lead to a *pass* verdict and a '1' at row i if the test run lead to a *fail* verdict.

The potentially incorrectly implemented blocks are then identified by calculating the similarity between the hit flags of every block, i.e., every column of the matrix, and the error vector. If the signature of a particular block is akin to the error vector, then this block is possibly incorrectly implemented.

Example 7.3. Table 7.1 shows the matrix and the error vector for the test runs shown in Figure 7.2. For example, when executing test case tc_2 on the specification of Figure 7.1, first block 4 is executed. The transition $ui!add$ causes the execution of block 2. Then the test case checks for the output of the implementation. The edges enabled in the test case execute the block 3 in the specification. Although the implementation opted for another output (different to error), we count block 3 as executed for tc_2. □

The similarity between an execution signature and an error vector is quantified by means of an similarity

$$\begin{array}{cc} & \text{Hit-Matrix} \\ & \leftarrow \text{n blocks} \rightarrow \\ \updownarrow\text{m runs} & \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1n} \\ h_{21} & h_{22} & \cdots & h_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{m1} & h_{m2} & \cdots & h_{mn} \end{bmatrix} \end{array} \qquad \begin{array}{c} \text{Error-} \\ \text{Vector} \\ \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix} \end{array}$$

Figure 7.3: Hit matrix summarizing the execution signatures of the different test runs and the error vector comprising the test execution results of these test runs.

Table 7.1: Binary hit matrix and the error vector for the six test runs of Figure 7.2.

Test Case	Blocks						Verdict
	1	2	3	4	5	6	
1	0	1	0	1	1	0	0
2	0	1	1	1	0	0	1
3	0	1	1	0	0	0	1
4	0	1	0	1	1	0	0
5	0	1	0	1	0	1	1
6	0	1	1	0	0	0	1

coefficient. While there are different similarity coefficient we use the Ochiai coefficient, which has been shown to perform well in practice (Abreu et al., 2007):

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}}$$

where $a_{pq}(j) = |\{i|x_{ij} = p \wedge e_i = q\}|$, and $p,q \in \{0,1\}$. $|X|$ denotes the cardinality of the set X, x_{ij} denotes the element at row i and column j of the binary hit matrix and e_i is the value of the error vector at row i. Informally, $a_{11}(j)$ denotes the number of rows where both the block j and the error vector have a one. On the contrary, $a_{10}(j)$ counts the rows where the block j has a one while the error vector has a zero.

Example 7.4. For the blocks of the matrix shown in Table 7.1 we have the following Ochiai coefficients: $s_O(1) = 0$, $s_O(2) = 0.817$, $s_O(3) = 0.866$, $s_O(4) = 0.5$, $s_O(5) = 0$, and $s_O(6) = 0.5$. □

Once the similarities between execution signatures and error vectors have been calculated, spectrum-based fault localization assumes that a high similarity indicates a high probability that the corresponding block of a program is the reason for the observed failure. As we apply spectrum-based fault localization on the level of a specification, a high similarity indicates a hight probability that the corresponding block has been implemented incorrectly.

7.2 Using Spectra to Group Test Cases

Our aim is to group test cases such that one only needs to analyze one test case per group in order to report on all detected discrepancies between the implementation and the specification. In this context, spectrum-based fault localization serves to calculate the failure probabilities, i.e. the Ochiai coefficients, of blocks of the specification. Then, we compute the similarity between all test cases with respect to these failure probabilities and put similar test cases into one group.

$$\begin{array}{cc}
 & \text{Hit-Matrix} \\
 & \leftarrow \text{l failed test runs} \rightarrow \\
\updownarrow \text{n blocks} & \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1l} \\ h_{21} & h_{22} & \cdots & h_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nl} \end{bmatrix}
\end{array}
\qquad
\begin{array}{c}
\text{Ochiai-} \\ \text{Vector} \\
\begin{bmatrix} s_O(1) \\ s_O(2) \\ \vdots \\ s_O(n) \end{bmatrix}
\end{array}$$

Figure 7.4: Similarity table used to calculate the similarities between test cases with respect to the Ochiai coefficient.

Table 7.2: Similarity table for our running example

Blocks	Test Run				s_O
	2	3	5	6	
1	0	0	0	0	0.000
2	1	1	1	1	0.817
3	1	1	0	1	0.866
4	1	0	1	0	0.500
5	0	0	0	0	0.000
6	0	0	1	0	0.500

The intuition behind using the failure probabilities is the following. Test cases having different execution signatures may have failed because they revealed the same misbehavior of the system. By weighting the execution signatures with the failure probabilities of the executed blocks, specification parts that are unlikely suspicious do not count that much on the similarity.

For the calculation of the similarities between the execution signatures of the failed test runs we build a matrix like that shown in Figure 7.4. In contrast to the hit matrix used in spectrum-based fault localization, the test runs build the columns; only failed test cases are added to the matrix. The blocks identified within the specification serve as rows. For each block we list its Ochiai coefficient in a separate vector.

Example 7.5. Table 7.2 shows the similarity table for our running example. The execution signatures of the failed test runs are entered into the columns. If the block i is executed by the test case j, then the entry at row i and column j is one, i.e. $x_{ij} = 1$; otherwise $x_{ij} = 0$. The last column of this table represents the Ochiai vector. □

The similarity between two test cases i and j, denoted by $s(i, j)$, is defined over the dissimilarity $ds_{tc}(i, j)$ of the test cases. That is, we count how many bits of the execution signatures of the test cases differ. More precisely, if there is a difference in the execution signature for block k, then we add the Ochiai coefficient to the dissimilarity value.

$$ds_{tc}(i, j) = \sum_{k=1}^{n} abs(x_{ki} - x_{kj}) * s_O(k)$$

In this formula n denotes the number of blocks identified in the specification. By the use of the failure likelihood, i.e. the Ochiai coefficient, blocks of the specification that are likely faulty count more on the dissimilarity than blocks that are unlikely faulty. Note that we normalize the value of $ds_{tc}(i, j)$ by dividing the result with the sum of all Ochiai coefficients, i.e.,

$$ds(i, j) = \frac{ds_{tc}(i, j)}{\sum_{k=1}^{n} s_O(k)}$$

Table 7.3: Similarity values for the four failed test cases of our running example.

	tc_3	tc_5	tc_6
tc_2	0.814	0.491	0.814
tc_3		0.491	1.0
tc_5			0.305

Algorithm 7.1 Test case grouping

```
1: $groups \leftarrow \emptyset$
2: while $failed \neq \emptyset$ do
3:     $t \leftarrow$ pick and remove one element from $failed$
4:     for all $group \in groups$ do
5:         if $\forall t' \in group \bullet s(t,t') > threshold$ then
6:             $group \leftarrow group \cup \{t\}$
7:             break
8:         end if
9:     end for
10:    if $t$ is in no group yet then
11:        $newgroup \leftarrow \{t\}$
12:        $groups \leftarrow groups \cup \{newgroup\}$
13:    end if
14: end while
15: return $groups$
```

Thus, the dissimilarity between two test cases i and j is a value between zero and one. The similarity is then simply obtained by subtracting $ds(i,j)$ from one:

$$s(i,j) = 1 - ds(i,j)$$

Example 7.6. The similarity measures of our four failed test cases of Figure 7.2 are depicted in Table 7.3. Note that the similarity is symmetric, i.e, $s(i,j) = s(j,i)$. For example, the similarity between the test runs of the test cases tc_2 and tc_3 is $s(tc_2,tc_3) = 0.814$. □

Two test cases are in the same group if they are similar. As it is infeasible to calculate a grouping such that the similarity between the test cases within each group takes a maximum we approximate the grouping. Algorithm 7.1 shows the pseudo code of our test case classification procedure. This algorithm takes a set of failed test cases ($failed$) and returns a partitioning of these test cases, i.e., no test case is in two groups and there are no test cases that are in no group.

Initially we start with an empty set of groups (Line 1). As long as there are failed test cases left (Line 2) we take one test case t and remove it from the set of failed test cases. Then we iterate through all groups (Line 4). If all test cases in a particular group are similar to t (Line 5), i.e., the similarity values of t and every other test case of the group are higher than a certain threshold, then we add t the this group (Line 6). A test case is only added to one group, i.e., we leave the loop after adding the test case to a group (Line 7). If we do not find a group for t (Line 10), then we generate a new group (Line 11) and update the set of groupings (Line 12).

The threshold is a parameter to our algorithm that decides which similarity value between two test cases denotes similar test cases. For experimental results that show the effects of this threshold on the grouping we refer to Section 8.8.3

Example 7.7. Given the set of failed test cases $failed = \{tc_2,tc_3,tc_5,tc_6\}$ and a similarity threshold value of 0.8, then the application of our classification algorithm works as follows: We pick one test case from

$failed$, e.g., tc_3. As there are no groups we immediately continue at Line 10 and $groups$ becomes $\{\{tc_3\}\}$. Then the algorithm continues at Line 3 and picks the next test case, e.g., tc_6. As there is one group in the set of groups, the body of the for-loop (Line 4) is executed once. When executing the for-loop's body the algorithm compares the similarity values of tc_6 an the test cases of the group, which is only tc_3 in this case. According to Table 7.3, the similarity value $s(tc_3,tc_6)$ is 1.0. As $1.0 > 0.8$ we add tc_6 to the existing group, getting an updated set of groups: $groups = \{\{tc_3,tc_6\}\}$.

Assume the algorithm picks tc_5 next (Line 3). When comparing similarity values between this test case and the test cases of the group $\{tc_3,tc_6\}$ we get $s(tc_3,tc_5) = 0.491$ and $s(tc_5,tc_6) = 0.305$. Therefore, the condition at Line 5 evaluates to false and tc_5 is not added to this group. As there is no other group the execution of the for-loop terminates and the algorithm continues at Line 10. tc_5 has not been added to a group yet, which causes the algorithm to generate a new group and to update the set of groups to $\{\{tc_5\},\{tc_3,tc_6\}\}$.

At this point tc_2 is the only one failed test case left. Evaluating the condition at Line 5 for tc_2 and the group $\{tc_5\}$ yields false as $s(tc_2,tc_5) = 0.491$ and the threshold is 0.8. Thus, in the next iteration of the loop at Line 4 the algorithm also evaluates the condition at Line 5 for tc_2 and $\{\{tc_3,tc_6\}\}$. As $s(tc_2,tc_3) = 0.814$ and $s(tc_2,tc_6) = 0.814$, tc_2 is added to this group (Line 6) and the loop is left (Line 7), i.e., $groups = \{\{tc_5\},\{tc_2,tc_3,tc_6\}\}$.

As there are no failed test cases left, the algorithm terminates. The final grouping says that tc_2, tc_3, and tc_6 detected the same failure while tc_5 detected a different failure. Comparing this result with the failure descriptions of the test cases in Example 2 shows that this grouping exactly resembles the detected failures. $\square$

Termination Algorithm 7.1 iterates as long as there are elements in the set of failed test cases, i.e. $failed \neq \emptyset$. The size of this set decreases continuously because in every iteration one element is removed at Line 3. Since no elements are added to this set, $failed$ eventually becomes empty and thus the algorithm terminates.

Runtime In the worst case we generate one group for each failed test case. In that case we have zero iterations of the loop at Line 4 for the first test case, one iteration for the second test case, and so on. In general, there are $(i-1)$ iterations of this for-loop for the i^{th} failed test case. Let n be the size of $failed$, i.e. the number of failed test cases, then we have a worst case complexity of

$$O\left(\sum_{i=1}^{n}(i-1)\right) = O\left(\frac{(n-1)*(n-1+1)}{2}\right) = O((n-1)*n) = O\left(n^2\right)$$

Thus, Algorithm 7.1 has a worst case asymptotic runtime of $O(n^2)$ where n is the number of failed test cases.

7.3 Blocks for LOTOS Specifications

One essential ingredient for spectrum-based fault localization and thus also for our approach is the granularity of blocks within a program or in our case the specification. In this section we briefly discuss possible partitions with respect to LOTOS specifications. Although this section is specific to the LOTOS specification language similar considerations apply to other specification languages as well.

Our choice of blocks is inspired by specification (code) coverage criteria. For measuring the coverage of test cases on a specification blocks in the specification are identified. If a block is hit by a test case then the block counts as covered. This is similar to what is needed by spectrum-based fault localization. We need to partition the specification into blocks.

There are different coverage criteria for LOTOS specifications (see Section 6.5). We use this coverage criteria to identify the blocks for LOTOS specification.

Processes

The coarsest partitioning is based on process coverage, which is similar to function coverage on imperative programs, and considers every process of a LOTOS specification as a block. In addition we create one block for the main behavior of the specification, i.e., the initial behavior after the behavior keyword which is not necessarily a process. If an action of a process's behavior is part of the test run, then the process is considered as executed.

Example 7.8. For the specification illustrated in Figure 7.1 this would mean that there are three blocks: The first block contains the main behavior at Line 3; The second block for the Main process extends from Line 5 to Line 27 and the third block for the DisplayStack process includes Line 29 to Line 35. □

Actions

The transitions of a test case are directly related to the actions of a LOTOS specification. Therefore, a reasonable partitioning of a LOTOS specification is to have one block per action statement. A block is marked as executed if the corresponding action is executed.

Example 7.9. The blocks with respect to the action statements are marked with gray color in Figure 7.1; there are 6 blocks with respect to this partitioning. Note that each action statement results in one or more transition in the underlying model of the LOTOS specification. A test case only needs to select one of these transitions in order to hit the action statement. □

Decisions

An action statement within a LOTOS specification results in multiple transitions within the underlying model, because data types are handled by enumeration. That is, every possible value combination of an action's parameters result in a single transition. Thus, using actions for partitioning a LOTOS specification can be refined by using logical statements over the specification's variables.

A LOTOS specification comprises guards which consist of logical decisions. Decision coverage is a commonly used criterion for assessing the quality of test suites. Thus, we propose to use the decisions of a specification for identifying the blocks of a LOTOS specification. As for the partitioning with respect to processes, we also use one block for the initial behavior of the specification. This is to capture possible failures related to an unguarded initial behavior.

Note that in the case of partitioning with respect to decisions we count each outcome of a decision as a distinct block. That is, the true outcome of a decision is a different block than the false outcome of a decision.

Example 7.10. Using decisions as partitioning criteria for the specification of Figure 7.1 results into eleven different blocks. One block for the main behavior of the process at Line 3. The Main process is split into six blocks by the outcomes of the three decisions at Line 9, Line 14, and Line 19, respectively. Finally, we have four blocks within the DisplayStack process, i.e., two blocks for the decision at Line 30 and another two blocks for the decision at Line 33. □

Conditions

Another commonly used coverage criterion is condition coverage Myers (1979). Conditions form a decision by composing them with logical operators (e.g., logical and, logical or). Inspired by condition coverage we use the conditions of a decision for a partitioning of LOTOS specifications. Thus, every outcome of a single condition serves as a separate block within a specification. Furthermore, similar to decision coverage, we add one separate block for the initial behavior of a LOTOS specification.

Example 7.11. Partitioning the specification illustrated in Figure 7.1 with respect to its conditions leads to 21 different blocks. First, there is the separate block for the initial behavior of the LOTOS specification, i.e., a block that is marked as hit if Line 3 is executed. Furthermore, we have one block for every truth value of every condition of the specification. For example the guard expression at Line 9 results in four blocks. `op eq display` gives two blocks, one for each truth value. The same is true for `size(s) gt 0`. When running test case tc_3 of Figure 7.2, then the following blocks are marked as executed. The block `(op eq display) = true` is executed. Furthermore, the block `(size(s) gt 0) = false` is also marked as hit. For the conditions of Line 14 we set the hit-flag for `(op eq add) = false` and for `(size(s) ge Succ(Succ(0))) = false`. Finally, for the four blocks derived from the conditions at Line 19 we get a hit for `(op eq add) = false`, `(size(s) lt Succ(Succ(0))) = false`, `(op eq display) = true`, and `(size(s) eq 0) = true`. As test case tc_3 does not execute the DisplayStack process we do not execute the blocks derived from the decisions of this process. □

Chapter 8

Case Study

This chapter summarizes the findings published in (Aichernig et al., 2007; Fraser et al., 2008b; Weiglhofer and Wotawa, 2008a; Weiglhofer et al., 2009a; Fraser et al., 2008a; Weiglhofer and Wotawa, 2008b; Weiglhofer et al., 2009,b; Peischl et al., 2007).

In this chapter we present the results obtained when applying the discussed techniques to two different protocols: (1) Session Initiation Protocol (SIP) (Rosenberg et al., 2002) and (2) Conference Protocol (CP) (Terpstra et al., 1996). For both protocols we derived test suites using the different test purpose based techniques of Chapter 6. Furthermore, we show the results obtained when running the generated test suites on real implementations of the two protocols. Finally, this chapter shows the results obtained by the use of our test case grouping approach of Chapter 7.

Table 8.1 summarizes the characteristics of the two specifications in terms of the number of processes, the number of actions (without `i` actions), the number of `i` actions, the number of decisions and conditions and in terms of net lines of code. We conducted all our experiments on a PC with Intel® Dual Core Processor 1.83GHz and 2GB RAM.

Table 8.1: Details of the used LOTOS specifications.

	No.pro-cesses	No.ac-tions	No.`i` actions	No.deci-sion	No. con-ditions	No.data-types	net. Lines of Code	
							Total	datatypes
SIP	10	27	8	39	49	20	3000	2500
CP	16	26	0	32	39	1	900	700

8.1 Conference Protocol

The Conference Protocol has been used previously to analyze the failure-detection ability of different formal testing approaches (e.g., du Bousquet et al. (2000); Belinfante et al. (1999)). The specification is available in different specification languages to the public[*]. In addition, there are 27 erroneous implementations which can be used to evaluate testing techniques.

The protocol itself is a simple communication protocol for a chat application. The main part of the application is called the Conference Protocol Entity (CPE). A CPE serves as chat client with two interfaces;

[*] http://fmt.cs.utwente.nl/ConfCase/

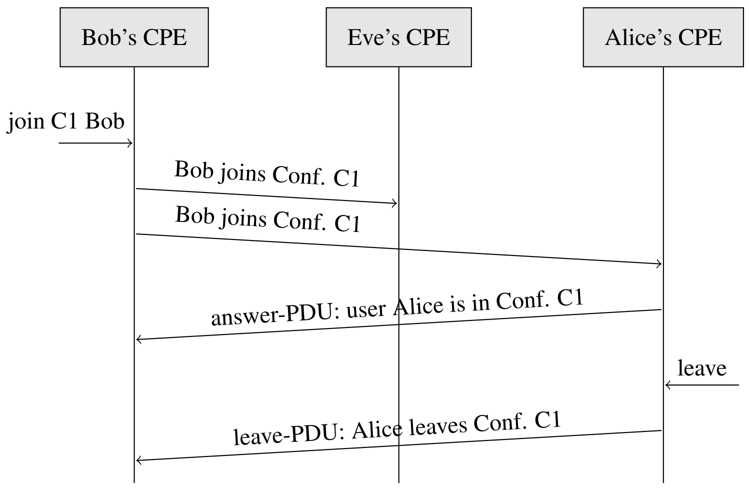


Figure 8.1: Simple call-flow illustrating joining and leaving a chat session.

one interface allows the user to enter commands and to receive messages sent by other users, and the other interface allows the chat application to send and receive messages via the network layer. These two interfaces are the points of control and observation for a tester.

Users can join conferences, exchange messages and leave conferences. Each user has a nick name and can only join one conference at a time.

Example 8.1. Figure 8.1 shows a typical example of a simple chat session. First, user Bob joins conference "C1" using the nickname Bob. The Conference Protocol entity sends that information to all potential conference partners. In the illustrated scenario user Alice participates in the same conference as joined by Bob. Thus, Alice's protocol entity answers with an *answer*-protocol data unit (PDU). Then Alice decides to leave the conference which causes her protocol entity to send a *leave*-PDU to all participating users, i.e., to Bob's CPE. □

8.2 Session Initiation Protocol

The Session Initiation Protocol is an RFC standard (Rosenberg et al., 2002) and handles communication sessions between two end points in the context of Voice-over-IP networks. *User management* comprises the determination of the location of the end system and the determination of the availability of the user. *Session management* includes the establishment of sessions, transfer of sessions, termination of sessions, and modification of session parameters. SIP defines various entities that are used within a SIP network. One of these entities is the *Registrar*, which is responsible for maintaining location information of users.

Example 8.2. An example call flow of the registration process is shown in Figure 8.2. In this example, Bob tries to register his current device as end point for his address Bob@home.com. Because the server needs authentication, it returns "401 Unauthorized". This message contains a digest which must be used to re-send the register request. The second request contains the digest and the user's authentication credentials, and the Registrar accepts it and answers with "200 OK". □

A full description of SIP can be found in the RFC 3261 (Rosenberg et al., 2002).

We developed a formal specification covering the full functionality of a SIP Registrar. Note that the Registrar determines response messages through evaluation of the request data fields rather than using different request messages. Thus, our specification heavily uses the concept of abstract data types.

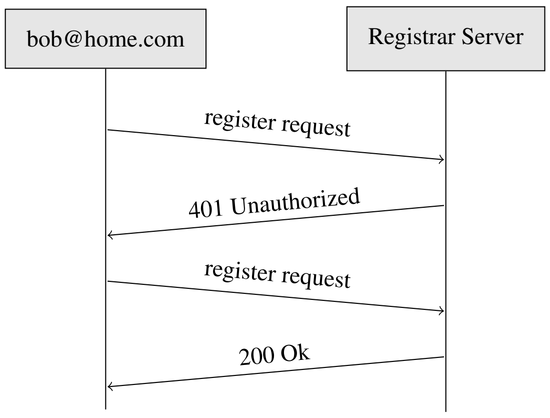


Figure 8.2: Simple call-flow of the registration process.

As system under test we used version 1.1.0 of the open source implementation OpenSER[†]. For some of our techniques we were also able to ran the generated test cases on the SIP Registrar developed by our project partner Kapsch CarrierCom. Unfortunately, Kapsch CarrierCom stopped the development of their SIP server and sold the software including the source code to another company. Thus, the commercial implementation was not available during the whole course of this thesis. However, whenever possible this section reports on the results obtained when applying the generated test cases to both the commercial and the open source implementation.

A Registrar does not necessarily need to authenticate users. According to the RFC a SIP Regsitrar SHOULD (Franks et al., 1999) authenticate the user. This is reflected in our specification by having a non-deterministic internal choice between using authentication and an unauthenticated mode. As both tested implementations allow one to turn authentication off, we ran all our test cases against two different configurations (with and without authentication).

8.2.1 Modeling the Session Initiation Protocol in LOTOS

In this section we present parts of our SIP Registrar specification (Weiglhofer, 2006) in order to exemplify the use of LOTOS.

A SIP Registrar provides its functionality through the maintenance of a state machine. Basically, this state machine is responsible for the retransmission of responses of the SIP Registrar. Figure 8.3 shows the so called non-invite server transaction of a SIP Registrar. We abstracted from nodes and edges drawn in gray since they are not relevant or hard to consider within a model suited for testing this protocol.

Each REGISTER request is processed by its own state machine. As illustrated by Figure 8.3, an initial request is handed over to the transaction's user (TU), i.e. the Registrar core. Based on the header fields of the request the Registrar core determines a proper response and forwards the response to the state machine. Responses in the Session Initiation Protocol are identified by three digits. Since a Registrar never generates `1xx` responses, our model does not include the gray edge from the *Trying*-state to the *Proceeding*-state. Any other response, i.e. `200-699`, is forwarded to the initiator of the request (*send response*). The state machine then goes to its *Completed*-state.

Once the *Completed*-state has been reached any request (which matches the transaction handled by this state machine) is answered with the last sent response (self-loop on the state *Completed*). After a particular amount of time, the state machine moves to the *Terminated*-state and the transaction is destroyed.

[†]`http://www.openser.org`; Note that in July 2008 OpenSER has been renamed to Kamailio and is now available at `http://www.kamailio.org/`

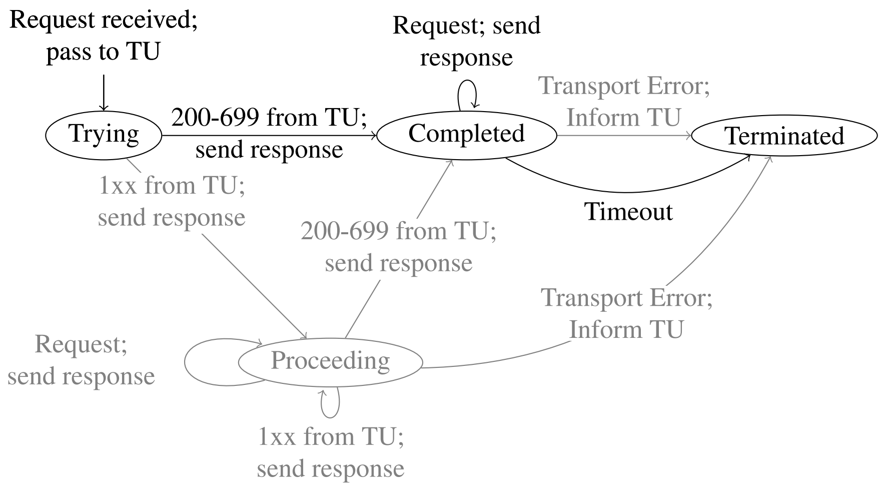

Request received; pass to TU
Trying
200-699 from TU; send response
Request; send response
Completed
Transport Error; Inform TU
Terminated
Timeout
1xx from TU; send response
200-699 from TU; send response
Transport Error; Inform TU
Request; send response
Proceeding
1xx from TU; send response

Figure 8.3: State machine for handling non-invite server transactions of a Session Initiation Protocol Registrar (Source: (Rosenberg et al., 2002)).

Figure 8.4 shows our LOTOS formalization of the transaction handling state machine. The state machine is represented by a single process (`serverTransaction`). This process communicates with the transaction user, i.e. the SIP Registrar core, through the gates `from_tu` and `to_tu`. The gates `pin` and `pout` are used for communication with the environment, i.e. to receive REGISTER requests and to send proper responses.

Since LOTOS does not include state-variables, the state machine is implemented as a recursive process where the current state is maintained in a parameter of this process. Every time the process is invoked it checks its current state and reacts according to Figure 8.3.

In addition to the current state parameter `trans_state` (Line 2), which may have the value `ts_trying`, `ts_completed`, or `ts_terminated`, the process takes two further parameters: `branch` (Line 3), which is used to identify retransmissions; `response` (Line 4), which holds the last sent response.

As indicated by the `noexit` keyword (Line 4), this server process does not terminate, but continues forever. Thus, as an abstraction we can only have one transaction state machine. The state machine is never destroyed, but only the relevant parameters are reset once the terminated state is entered.

If the model of the state machine is in the *Terminated*-state, it waits to receive a message and moves on to the *Trying* state if a message is received. The reception of REGISTER messages (and some initial validations) is modeled by the process `listenForMessage` (Line 7). On successful termination the process `listenForMessage` passes the control to the succeeding behavioral block (» operator). If a valid message has been received, indicated by the `hand_to_tu` variable, then the request is forwarded to the transaction user (Line 11) and the state machine moves to the *Trying*-state (Line 13).

If the state machine is in the *Trying*-state (Line 23), any response from the transaction user (Line 24) is sent to the environment of the Registrar (Line 25). After that the state machine moves on to the *Completed*-state (Line 27).

In the *Completed*-state there is a non-deterministic internal choice between receiving a retransmission (Lines 32-36) and finally moving to the *Terminated*-state (Line 42). This non-deterministic choice is an abstraction for the timeout transition in the state machine.

8.2.2 Simplifications

As one can already see from the LOTOS model of the non-invite server transaction, modeling always includes choosing proper simplifications. Although modern specification languages have high expressive

```
process serverTransaction [pin, pout, from_tu, to_tu](
  trans_state: TransState,
  branch: Branch,
  response: SipResp) : noexit :=

  [trans_state eq ts_terminated] -> (
    listenForMessage[pin,pout,from_tu,to_tu](branch,response) >>
    accept msg: RegisterMsg, resp: SipResp, hand_to_tu: Bool in
    (
      [hand_to_tu] -> (
        to_tu !msg;
        serverTransaction[pin, pout, from_tu, to_tu](
          ts_trying, getBranch(msg) + 1, resp)
      )
      []
      [not(hand_to_tu)] -> (
        serverTransaction[pin, pout, from_tu, to_tu](
          ts_terminated, 0, resp)
      )
    )
  )
  []
  [trans_state eq ts_trying] -> (
    from_tu ?resp: SipResp;
    pout !resp;
    serverTransaction[pin, pout, from_tu, to_tu](
      ts_completed, branch, resp)
  )
  []
  [trans_state eq ts_completed] -> (
    (
      i;
      pin ?msg: RegisterMsg [(getBranch(msg) eq branch)];
      pout !response;
      serverTransaction[pin, pout, from_tu, to_tu](
        ts_completed, branch + 1, response)
    )
    []
    (
      i;
      serverTransaction[pin, pout, from_tu, to_tu](
        ts_terminated, 0, response)
    )
  )
endproc
```

Figure 8.4: LOTOS specification of the transaction handling state machine of a Session Initiation Protocol Registrar.

power, it is impractical and often infeasible to model the complete concrete behavior of a system. Thus, when developing a formal model one usually abstracts from the real world.

Basically we distinguish between two different types of simplifications: *abstractions* and *limitations*. Abstractions are simplifications that preserve conformance. For example, one may only specify the behavior for a particular set of inputs; for the unspecified inputs the systems may behave arbitrarily. Hence, in the context of **ioco** with partial models, abstraction may constrain the inputs and may remove constraints from the output behavior. In contrast, a limitation is a restriction of the system's possible reactive behavior (output) and hence not a proper abstraction. Consequently, limitations do not preserve conformance and the tester must be careful in interpreting a fail verdict: it might be due to a limitation in the model.

In model-based testing the simplifications influence the kind of detectable failures. The more abstract or limited a formal model is, the less information for judging on the correctness of an implementation is available (Gaudel, 1995). Thus, a major challenge in deriving models for industrial applications is the se-

Table 8.2: Simplifications for the specification of the SIP Registrar.

id	type	description
1	limitation	Our formal model of the Registrar never terminates with a server error.
2	limitation	Our specification never forwards REGISTER messages to other SIP Registrars.
3	limitation	We assume that the communication channel is reliable and delivers messages in the sent order.
4	functional	The Registrar starts from a well known initial state.
5	functional	While the authentication handshake is in our model, the calculation of authentication credentials is not modeled.
6	functional	REGISTER messages do not contain any REQUIRES header fields.
7	data	The CALL-ID is abstracted to the range $[0,1]$.
8	data	We limit the integer part of the CSEQ header to $[0,1]$. The method part is not in the formal model.
9	data	The range $[0,2^{32-1}]$ of the EXPIRES header field can be divided into three partitions where we use only boundary values of each partition.
10	data	Our model uses three different users: An authorized user, a known but unauthorized user and an unknown user.
11	data	Our formal model uses three different CONTACT values: *, any_addr1, and any_addr2.
12	data	The TO and FROM header fields are omitted in our abstract REGISTER messages.
13	temporal	Our specification does not use any timers. We only focus on the ordering of events.

lection of proper simplifications. Simplifications need to limit a specification's state space to a manageable size, while the model still needs to be concrete enough to be useful.

According to Prenninger and Pretschner (2005) and to Pretschner et al. (2005) we distinguish five classes of abstractions: *functional*, *data*, *communication*, *temporal*, and *structural* abstractions. *Functional* abstraction focuses on the functional part of the specification. This class of abstractions comprises the omission of behavior that is not required by the objectives of the model. *Data* abstraction subsumes the mapping from concrete to abstract values. Data abstraction includes the elimination of data values that are not needed within the functional part of the specification. *Communication* abstraction maps complex interactions at a more abstract level, e.g., the formal model uses one message to abstract a handshake scenario (several messages) of the real world. *Temporal* abstraction deals with the reduction of timing dependencies within the formal specification. For example, a certain model specifies only the ordering of events, but abstracts from discrete time values. *Structural* abstraction combines different real world aspects into logical units within the model.

Simplifications for the SIP Registrar Model

When developing the formal model of a SIP Registrar we have chosen the simplifications listed in Table 8.2.

In particular, we simplify our model with respect to general server errors (Simplification 1) because of the loose informal specification of server errors within the RFC. Server errors may occur at any time when the Registrar encounters an internal error. For testing general server errors we would need a significant knowledge about the implementation internals. Especially, in order to trigger a server error during test execution, we would need to know how to enforce it. Hence, we skip server errors from the modeled behavior which may result in a wrong testing verdict. Therefore, this simplification is a limitation.

Simplification 2 omits specification details about forwarding requests. Thus, we do not generate tests for this feature. This is again a limitation as the forwarding of requests would result in different outputs, i.e. the receiver of the forwarded request responds to the REGISTER message.

Simplification 3 removes the needs for modeling possible interleaving of messages. During test execution this assumption is ensured by running the test execution framework and the implementation under test on the same computer.

Simplification 4 requires the start from a defined initial state. Otherwise, our model would have to consider different database contents on startup of the Registrar. We consider this a functional abstraction, because the functionality for other initializations is left open.

Simplifying the model with respect to the calculation of authentication credentials (Simplification 5) does not impose any limitation if the credentials are calculated and inserted correctly into test messages during test execution. As the detailed algorithm for credentials calculation is abstracted, this is a functional abstraction.

We also skipped the REQUIRES header field in the formal specification in order to limit the number of possible request messages (Simplification 6). Not considering this input header field as being part of a REGISTER request represents a functional abstraction: the implementation may behave arbitrarily after this unspecified request.

Simplifications 7-10 are based on the ideas of equivalence partitioning and boundary value analysis Myers (1979), which are strategies from white-box testing. For example, Simplification 10 uses the fact that the Registrar relevant part of the RFC only distinguishes users that (1) are known by the proxy and allowed to modify contact information, (2) that are known by the proxy but are not allowed to modify contact information, and (3) users that are not known by the proxy, i.e. users that do not have an account. Thus, only three different users are needed, one from each group. Note that the simplifications 7-10 are data abstractions. Each of the header fields addressed by these abstractions are inputs to our specification. However, as we have one value per equivalence partition this is not a functional abstraction: every behavior (with respect to these inputs) of the system is modeled. Nevertheless, as we do not model all possible input values this can be seen as a data abstraction.

Simplification 11 limits the different CONTACT header field values. We allow the two regular addresses with the abstract values "any_addr1" and "any_addr2", respectively. These two elements are replaced during test execution with valid contact addresses. According to the RFC, the asterisk is used for "delete" requests. This is again a data abstraction as the different possible behaviors are covered by our specification.

Simplification 2 causes the header fields, TO and FROM, to contain redundant information. So they can be omitted from our formal REGISTER messages (Simplification 12). Again this is a data abstraction.

Finally, as we do not consider testing under the presence of timing constraints, we abstract from concrete time events (Simplification 13).

8.3 Abstract Test Case Execution

Due to simplifications formal models are usually more abstract than real implementations. Consequently, the abstract test cases derived from such models cannot be directly applied to the implementation under test. To adapt abstract test cases to the IUT it is necessary to reinsert the information abstracted within the model into the test case. More precisely, every stimuli i has to be converted to a concrete message $\gamma(i)$, while a system response o has to be mapped to the abstract level $\alpha(o)$ (Pretschner and Philipps, 2005).

Basically, the execution of abstract test cases consists of two different tasks. One task is test control, i.e., select the next test message and determine the test verdict. The other task is the transformation of the abstract test messages to concrete stimuli and the conversion of concrete system responses to abstract test messages.

Algorithm 8.1 Test control algorithm for a test case $t = (Q, L_U \cup L_I, q_0, \rightarrow)$.

```
1:  $q_c \leftarrow q_0$
2:  while $q_c \notin \{fail, pass, inconclusive\}$ do
3:      if $\exists q' \in Q, o \in L_U : q_c \xrightarrow{o} q'$ then
4:          send(o)
5:          $q_c \leftarrow q'$
6:      else
7:          $msg \leftarrow receive()$
8:          if no input for particular amount of time then
9:              $q_c \leftarrow q_c$ after $\theta$
10:         else
11:             $q_c \leftarrow q_c$ after $msg$
12:         end if
13:     end if
14: end while
15: return $q_c$
```

8.3.1 Test Control

The test control procedure for an abstract test case, which is given in terms of an LTS with inputs and outputs, is illustrated by Algorithm 8.1. This algorithm takes a test case TC and returns a verdict when executing the test case against an implementation under test (IUT). The two methods `send(message)` (line 4) and `receive()` (line 7) denote the communication with the IUT. These two methods encapsulate the concretization of requests and the abstraction of responses, respectively. Starting at the initial state q_0 (line 1) of the test case our test execution algorithm traverses the test case until the current state q_c is a verdict state (line 2). If the current state q_c has an output-labeled edge (line 3), the stimuli which is represented by the label of the edge is sent to the IUT (line 4). Otherwise, the algorithm waits for a response of the IUT (line 7). If there is no input from the implementation under test for a particular amount of time, then the implementation is quiescent. In other words, a timeout of a receive request denotes the reception of a δ event. The observation of this timeout generates a θ event (line 9). In case the algorithm receives a response, it updates the current state q_c (line 11).

Note that the controllability property (see Section 3.1) of test cases simplifies the test control algorithm, because the algorithm does not need to handle inputs and outputs at the same state. Furthermore, as test cases are deterministic, q_c **after** *msg* always returns a single state. As test cases are required to have a finite behavior, this algorithm eventually terminates when a verdict state is reached.

As we rely on TGV for test case generation the obtained test cases are not input enabled, i.e. our test cases may require to prevent an implementation from doing outputs (see Chapter 5).

However, in practice it is often not possible that a tester prevents an implementation from doing outputs. One way to overcome this issue is to apply the *reasonable environment assumption* (Fernandez et al., 1997), which says that before the environment sends a message to the network, it waits until stabilization. This means that the test execution environment is not allowed to send new messages until it has received all responses from the implementation. In order words, one gives outputs (of the IUT) priority over inputs (to the IUT).

Recall that TGV prunes edges during test case generation, as this tool relies on blocking outputs from the IUT while inputs are enabled. Thus, we implement the reasonable environment assumption by running the specification parallel to the test case execution. If there is an input from the implementation to a test case and this input is allowed by the specification but not by the test case we give an inconclusive verdict. If the input is not allowed by both the test case and the specification we give a fail verdict. Otherwise, i.e. the input is allowed by the test case, we continue the execution of the test case.

Note that the reasonable environment assumption is not a general assumption of our approaches. We only use it to ensure correct verdicts during test case execution. If the assumptions of **ioco** are satisfied then there is no need for using the reasonable environment assumption.

8.3.2 Message Transformation

For the transformation of messages, we use a simple rule-based rewriting system. The rewriting system comprises two sets of rules that define how to transform abstract into concrete messages and vice versa. Furthermore, our rewriting system uses a set of named mappings that associates concrete and abstract values. Global variables can be used to convey information through different rewritten messages. For example, such variables are used to store authentication data which is only given in a received message but which is needed during the rewriting of the next message.

A rewriting rule is a binary function that takes the abstract or the concrete message as first parameter and the current intermediate result from previous applied rules as second parameter. The rewriting rule returns the transformed message.

Currently we use nine different types of rules that are listed in Table 8.3. The definition of the rules uses the following notation: $\phi(regex, text)$ is a boolean predicate which returns true if the regular expression *regex* matches in $text$. The letter i denotes the start-index of the match, while ℓ denotes the end-index of the match. $s\widehat{\ }t$ stands for the concatenation of the two elements s and t. $e[j:k]$ denotes the part from index j to index k of an element e. $\mathcal{V}_S(var)$ gives the value of a variable var, while $\mathcal{V}_S(var) := x$ assigns the value x to variable var. $\mathcal{M}_S(y).abs(C,i)$ denotes the retrieval of the i-th abstract value for the mapping y given the list of concrete values C. $\mathcal{M}_S(y).con(A,j)$ gives the j-th concrete value for the list of abstract values A of mapping y.

Example 8.3. We will use a simple registration protocol for the illustration of our approach. By the use of this protocol users can register at a server or users can remove all registrations. The protocol comprises three different message types for registration, removal of all registrations and command confirmation. A registration command consists of the prefix "reg:" and the user name (e.g., reg:foo;). Registrations are removed with the "clr;" command. The registration command is confirmed with the prefix "ok:" followed by the name of the registered user. The response to the registration removal command is also "ok:" followed by a semicolon separated list of user names. The list comprises the names of the users that had been removed. For example a session between the server S and the client C may look like the following:

$$C \xrightarrow{reg:foo;} S; S \xrightarrow{ok:foo;} C; C \xrightarrow{reg:bar;} S; S \xrightarrow{ok:bar;} C; C \xrightarrow{clr;} S; S \xrightarrow{ok:foo;bar;} C$$

The abstract test cases use two different users which are represented by the abstract values 0 and 1. Thus, a test case may use one of the following test messages, where user is replaced by 0 or 1: *I !reg !user*, *I !clr*, and *O !add(user,nil)* or *O !add(user,add(user,nil))*. Using these abstract messages, the example session from above is represented as follows: *I !reg !0*, *O !add(0,nil)*, *I !reg !1*, *O !add(1,nil)*, *I !clr*, *O !add(0,add(1,nil))*.

Figure 8.5 shows the rule system for that simple protocol. Note that this rule system is designed for testing the server side implementation of the protocol. Thus, it only contains rules for refining *I !reg !user*, *I !clr* and for abstracting *O !add(...)*.

Refining the first message *I !reg !0* from the sequence above comprises the following steps: The first refinement rule $if^{R1}_{.*clr}$ checks if the messages is a clear request. Since ".*clr" does not match in the message, the rules of $R1$ are not applied. Because ".*reg" matches in the abstract message, we continue with the rules of $R2$. The first rule of $R2$ appends "reg:" to the intermediate result. In the next step app^{R3} is evaluated. Since, $R4$ returns 0 (the sub expression of interest is marked with brackets in rule $R4$), $R3$ returns the concrete value of the mapping "users" for 0 which is "foo". Thus, "foo" is appended to the intermediate result which is now "reg:foo". Finally, the last rule of $R2$ appends a semicolon leading to the result: "reg:foo;". □

rule	description
app.	Append the result of the list of rules R to s. $app^R(m,s) = s\frown R(m,s)$
substi-tute	Given a regular expression c, this rule replaces the matched text within s by the result of R. $sub_c^R(m,s) = \begin{cases} s[0:i]\frown R(m,s)\frown sub_{c,t}^R(m, s[\ell : length(s)-1]) & \text{if } \phi(c,s) \\ s & \text{otherwise} \end{cases}$
condi-tion	Apply a list of rules R if the regular expression c matches within the element m. $if_c^R(m,s) = \begin{cases} R(m,s) & \text{if } \phi(c,m) \\ s & \text{otherwise} \end{cases}$
loop	Apply a list of rules R for every match of a regular expression c in the message m. $for_c^R(m,s) = \begin{cases} for_c^R(m[\ell : length(m)-1], R(m,s)) & \text{if } \phi(c,m) \\ s & \text{otherwise} \end{cases}$
save	Save the result of R at location x. $sav_x^R(m,s) = \mathcal{V}_S(x) := R(m,s)$
load	Retrieve the value stored in x. $val_x(m,s) = \mathcal{V}_S(x)$
map-ping	Map abstract values to a concrete value and vice versa. $con_{y,j}^R(m,s) = \mathcal{M}_S(y).con(R(m,s), j)$ $abs_{x,i}^R(m,s) = \mathcal{M}_S(x).abs(R(m,s), i)$
part	Extracts a matching regular (sub) expression r from m $part_r(m,s) = \begin{cases} m[i:\ell] & \text{if } \phi(r,m) \\ \textit{empty element} & \text{otherwise} \end{cases}$
text	Returns the fixed text t $text_t(m,s) = t$

Table 8.3: Rules for the specification of abstraction and refinement functions; each rules takes the abstract/concrete message m as first parameter, and the intermediate result s as second parameter. The rules return the result of the transformation.

Our rewriting rules of Table 8.3 are suitable for handling simplifications where simple abstract values are mapped to concrete values, e.g., the abstract values 0, 1 are mapped to concrete user names “foo” and “bar”. For simplifications that omit parts of concrete messages (e.g. Simplification 12 of Table 8.2: Omitted header fields in SIP messages) our rewriting system can be used to insert a constant hard-coded value into the concrete message. Also the formatting of the message, e.g. adding or removing spaces, brackets and other control characters, can be handled by our rewriting system.

However, for more complex simplifications (e.g. Simplification 5 of Table 8.2: we do not model the calculation of authentication credentials) the rewriting system has to be extended.

Thus, in our implementation we have an additional rule which calculates the authentication credentials with respect to the algorithms described by Franks et al. (1999). Furthermore, our implementation of the test control algorithm allows one to communicate with an implementation under test through TCP, UDP or the standard input-output streams of unix processes.

We use our execution framework for the execution of abstract test cases for both, the SIP Registrar and the Conference Protocol. For the SIP Registrar, the rule sets for γ and α comprise 66 (12 app^R, 9 sub_c^R, 7 if_c^R, 1 for_c^R, 5 val_x, 11 $con_{y,j}^R$, 9 $part_r$, and 12 $text_t$) and 36 (10 app^R, 5 if_c^R, 4 for_c^R, 3 sav_x^R, 1 $abs_{x,i}^R$,

Mappings:	$usr:(\{0,foo\},\{1,bar\})$		
Variables:	$\emptyset$		
Rules:	$\gamma=\left(if^{R1}_{.*clr}, if^{R2}_{.*reg}\right)$, $\alpha=\left(text_{ok:}, for^{R5}_{add(}\right)$		
	$R1=(text_{clr;})$	$R2=\left(text_{reg:}, app^{R3}, app^{(text;)}\right)$	$R3=\left(con^{R4}_{usr,0}\right)$
	$R4=\left(part_{.*!([0-9]+)}\right)$	$R5=\left(abs^{R6}_{usr,0}, app^{(text;)}\right)$	$R6=\left(part_{([A-Za-z]*)}\right)$

Figure 8.5: Rule system $\mathcal{R}_S$ for the simple registration protocol

3 $part_r$, and 10 $text_t$) rules, respectively. We use 4 mappings and 3 variables. On the contrary, in the case of the Conference Protocol γ comprises 54 (19 app^R, 9 if^R_c, 11 $con^R_{y,j}$, and 15 $text_t$) rules, while α consists of 60 (27 app^R, 6 if^R_c, 9 $con^R_{y,j}$, and 18 $text_t$) rules. The Conference Protocol uses 3 mappings.

8.4 Multiple Test Cases per Test Purpose

This section summarizes the improvements gained from extracting multiple test cases for a single test purpose (see Section 6.3). Therefore, we manually developed 10 test purposes for the LOTOS version of the Conference Protocol specification. For the Session Initiation Protocol we identified five relevant scenarios from the textual specification. We formalized these scenarios in terms of test purposes.

8.4.1 Test Case Generation Results

Tables 8.4 and 8.5 show the results in terms of the numbers of test cases generated with the discussed methods: A single test case per test purpose, state coverage (S), input-label coverage (IL), output-label coverage (OL), (all) label coverage (AL), and transition coverage (T). Here, inputs and outputs are given in terms of test cases, i.e., inputs are responses from implementations while outputs are stimuli to be sent to implementations. The Tables 8.6 and 8.7 show for each manually generated test purpose the time needed to derive the number of test cases stated in Table 8.4 and in Table 8.5. The left parts of these tables show the results using all test cases generated by the observer based approach, i.e., in that case we did not check for already covered observers. In contrast to that the right parts show the results when only using test cases that covered new observers as described in Section 6.3.1.

Minimizing the generated test suites for the Conference Protocol with respect to the covered observers reduced the number of test cases in the state coverage based test suite by approximately 61%. For input label, output label, and (all) label coverage we got a reduction of approximately 50%, 29%, and 51%, respectively. The size of the test suite for transition coverage was reduced by approximately 54%. In the case of the SIP Registrar we got a reduction of approximately 44%, 46%, 3%, 4%, and 12% for the different types of coverage criteria.

Tables 8.6 and 8.7 illustrate the performance of our prototype implementation of the presented algorithm. Using multiple test cases per test purpose, the complete test graphs generated by TGV were split into test cases within reasonable time. Note that the time needed by TGV for generating a complete test graph is higher than the time needed to generate a single test case. These times are not included in the figures listed in the Tables 8.6 and 8.7. The time needed to generate the complete test graphs are listed in the Tables 8.8 and 8.9.

Tables 8.8 and 8.9 list details of the complete test graphs extracted for the different test purposes including the average numbers (∅) for all test graphs. The tables list for each test purpose (1st column), the time needed (2nd column) by TGV to extract the complete test graph from the specification. The third column shows the total number of states while the fourth and the fifth column contain the number of pass-verdict

Table 8.4: Number of test cases for the Conference Protocol.

TP	Single	Regular					Minimized				
		S	IL	OL	AL	T	S	IL	OL	AL	T
1	1	48	22	33	55	239	21	17	28	36	167
2	1	32	20	25	45	111	22	17	20	28	72
3	1	13	5	6	11	17	2	2	2	3	4
4	1	16	8	17	25	69	7	4	13	14	26
5	1	20	6	20	26	36	4	2	15	15	4
6	1	17	6	19	25	55	2	1	13	14	5
7	1	22	8	18	26	61	2	2	11	11	8
8	1	42	26	19	45	103	15	9	13	12	43
9	1	59	20	13	33	133	27	5	4	8	60
10	1	25	20	17	37	61	10	11	14	20	19
Σ	10	294	141	187	328	885	112	70	133	161	408

Table 8.5: Number of test cases for the Session Initiation Protocol.

TP	Single	Regular					Minimized				
		S	IL	OL	AL	T	S	IL	OL	AL	T
1	1	33	18	306	324	2174	18	4	302	302	2098
2	1	12	16	1116	1132	1145	11	11	1116	1116	1116
3	1	507	24	1728	1752	2764	280	17	1593	1593	1943
4	1	2	2	660	662	662	1	1	660	660	660
5	1	4	3	996	999	1002	3	1	996	996	996
Σ	5	558	63	4806	4869	7747	313	34	4667	4667	6813

states and inconclusive-verdict states, respectively. In addition, Tables 8.8 and 8.9 contain the number of input labels, output labels as well as the total number of labels. Column nine and ten contain the average and the maximal branching factor. The branching factor denotes the number of outgoing edges for a particular state. Finally, column eleven shows the number of transitions of the complete test graphs.

8.4.2 Test Case Execution Results

We executed the generated test cases on our implementations under test. For the Conference Protocol we used the 27 different implementations provided with the specification. For our second specification the OpenSER implementation of the SIP Registrar served as system under test.

The two Tables 8.10 and 8.11 show the obtained results in terms of the number of obtained pass verdicts (3rd and 8th column), fail verdicts (4th and 9th column) and inconclusive verdicts (5th and 10th column) for each coverage criterion (1st column). In addition, these tables show the number of executed test cases (2nd and 7th column) and the number of detected failures (6th and 11th column). Both tables list the results for the regular (2nd to 6th column) as well as for the minimized test suites (7th to 11th column).

As these two tables show, there was no observable degradation of the failure sensitivity through the minimization. To further assess the quality of the generated test cases Table 8.12 shows the function coverage and condition/decision coverage on our implementations under test. The results listed in this table apply to the full test suites as well as to the minimized test suites; i.e, there was also no observable degradation of the code coverage through the minimization.

Table 8.6: Creation times for the Conference Protocol.

TP	Single	Regular					Minimized				
		S	IL	OL	AL	T	S	IL	OL	AL	T
1	1s	3s	<1s	2s	3s	18s	2s	<1s	2s	2s	31s
2	1s	1s	<1s	1s	2s	8s	1s	<1s	1m	2s	7s
3	<1s	<1s	<1s	<1s	<1s	<1s	<1s	<1s	<1s	<1s	<1s
4	5s	<1s	<1s	<1s	1s	2s	<1s	<1s	<1s	<1s	2s
5	1s	2s	<1s	1s	2s	8s	2s	<1s	1s	1s	7s
6	2s	<1s	<1s	1s	1s	5s	<1s	<1s	1s	1s	4s
7	2s	16s	<1s	25s	26s	3m53s	17s	<1s	25s	26s	3m52s
8	<1s	2s	<1s	<1s	2s	10s	1s	<1s	<1s	1s	9s
9	<1s	3s	<1s	<1s	<1s	39s	2s	<1s	<1s	<1s	37s
10	4s	<1s	<1s	<1s	<1s	2s	<1s	<1s	<1s	<1s	2s
Σ	19s	31s	10s	35s	40m	5m26s	29s	10s	35s	37s	5m32s

Table 8.7: Creation times for the Session Initiation Protocol.

TP	Single	Regular					Minimized				
		S	IL	OL	AL	T	S	IL	OL	AL	T
1	24s	8s	5s	24s	19s	16m13s	8s	6s	24s	23s	19m41s
2	5s	<1s	2s	1m36s	1m20s	1m20s	<1s	1s	1m31s	2m01s	2m00s
3	3s	56s	5s	48s	5m20s	1m27s	54s	4s	41s	6m15s	11m29s
4	6s	<1s	<1s	2m13s	28s	27s	<1s	<1s	1m57s	41s	41s
5	5s	<1s	<1s	8m10s	1m00s	1m00s	<1s	<1s	7m37s	1m33s	1m33s
Σ	43s	1m07s	14s	13m11s	8m27s	20m27s	1m05s	13s	11m29s	10m53s	35m24s

8.4.3 Discussion

This section summarizes our findings when applying the approach presented in Section 6.3 to two different applications. The middle-sized conference protocol specification (approx. 900 LOC) comes with 27 faulty implementations. Theoretically 25 faulty implementations can be detected using input-output conformance testing. The second specification is a formalization of a real-world protocol. Therefore we were able to apply the generated test cases to the open-source implementation OpenSER.

Test Purposes. For both specifications we manually derived test purposes. The results obtained for Conference Protocol using our test purposes clearly indicate that writing good test purposes is not trivial. Testing this protocol using test purposes was subject to previous work by du Bousquet et al. (2000), where also ten test purposes have been used. However, du Bousquet et al. were able to detect 24 of the 25 non-conforming mutants. Unfortunately, their test purposes were not available for us, and our test purposes only detected 23 erroneous implementations in the best case.

Minimization. Interestingly, the SIP Registrar test suites were less redundant (in terms of covered observers) than the Conference Protocol. This is because the complete test graphs for the Registrar heavily branch. The average branching factor of the complete test graphs for the SIP Registrar is 115.7 edges (Table 8.9) while this value is 2.79 for the Conference Protocol (Table 8.8). This is because different stimuli (outputs of a test case) may cause the same response message from a Registrar (inputs to a test case). For example, malformed requests are always rejected by an *invalid request* response from the server. Thus, we

Table 8.8: Characteristic figures for the complete test graphs of the Conference Protocol.

TP	time	states			labels			branching		tran-sitions
		total	pass	inconc.	input	output	total	avg.	max.	
1	16m36s	49	18	0	22	33	55	4.88	16	239
2	26s	33	18	0	20	25	45	3.36	16	111
3	1s	14	3	0	5	6	11	1.21	2	17
4	3m24s	17	4	0	8	17	25	4.06	16	69
5	3s	21	2	1	6	20	26	1.71	8	36
6	3m46s	18	1	0	6	19	25	3.06	9	55
7	3m36s	23	2	0	8	18	26	2.65	9	61
8	4m13s	43	14	0	26	19	45	2.40	8	103
9	2m12s	60	8	1	20	13	33	2.22	8	133
10	11m07s	26	12	0	20	17	37	2.35	12	61
∅	4m32s	30.4	8.2	0.2	14.1	18.7	32.8	2.79	10.4	88.5

Table 8.9: Characteristic figures for the complete test graphs of the Session Initiation Protocol.

TP	time	states			labels			branching		tran-sitions
		total	pass	inconc.	input	output	total	avg.	max.	
1	3m48s	34	9	2	18	306	324	63.94	300	2174
2	5s	13	14	2	16	1116	1132	88.08	1116	1145
3	19s	508	12	11	24	1728	1752	5.44	864	2764
4	5s	3	1	1	2	660	662	220.67	660	662
5	6s	5	1	2	3	996	999	200.40	996	1002
∅	53s	112.6	7.4	3.6	12.6	961.2	973.8	115.7	787.2	1549.4

needed different test cases to cover the different possible stimuli. In contrast to that the structure of the Conference Protocol specification allowed more minimization.

Test Suite Quality. The code coverage (Table 8.12) and the number of detected failures (Tables 8.10 and 8.11), show the improvement gained from using multiple test cases per test purpose. Using a single test case per test purpose 17 erroneous implementations of the Conference Protocol and 1 failure within the OpenSER Registrar have been detected. Although, at least in the case of the Conference Protocol this is better than we expected and significant improvement is possible. Using more test cases per test purpose revealed more actual failures than using only one test case per test purpose (in the best case 23 erroneous implementations for the Conference Protocol and 6 instead of 2 failures for the SIP Registrar).

8.5 Mutation-based Test Purposes

This section presents the results obtained when applying mutation-based test purposes, as discussed in Section 6.4, to our two applications under test. We report on the results obtained when using both of the presented approaches for handling large state spaces, i.e. bounded conformance and slicing-via-test-purpose.

Table 8.10: Test case execution results for the Conference Protocol

Test Suite	Regular					Minimized				
	no.tc.	pass	fail	inconc.	failures	no.tc.	pass	fail	inconc.	failures
Single	10	195	75	0	17	10	195	75	0	17
S	294	4994	1891	1053	23	112	787	1886	351	23
IL	141	2609	862	336	18	70	1398	425	67	18
OL	187	3516	1263	270	21	133	2629	875	87	21
AL	328	6143	2099	614	21	161	3175	1038	134	21
T	885	15972	5816	2107	23	408	7610	2500	906	23

Table 8.11: Test case execution results for the Session Initiation Protocol Registrar

Test Suite	Regular					Minimized				
	no.tc.	pass	fail	inconc.	failures	no.tc.	pass	fail	inconc.	failures
Single	5	5	1	4	1	5	5	1	4	1
S	558	199	412	505	5	313	114	229	283	5
IL	63	31	69	26	4	34	19	17	32	4
OL	4806	2383	5250	1979	5	4667	2906	3222	3206	5
AL	4869	2414	5319	2005	5	4667	2906	3222	3206	5
T	7747	4507	4814	6173	6	6813	4160	4147	5319	6

8.5.1 Test Case Generation Results

We developed a mutation tool that takes a LOTOS specification and uses the mutation operators of Section 6.4.2 in order to generate for each possible mutation one faulty version (mutant) of the specification. With this tool all mutants of the specifications were generated automatically. Furthermore, our mutation tool allows one to insert the markers for our *slicing via test purpose* technique (see Section 6.4.4).

Bounded Input-Output Conformance

Table 8.13 lists the results obtained when using bounded input-output conformance. This table lists for each mutation operator (1st column) the overall number of generated mutants (2nd and 7th column) for the Session Initiation Protocol specification and for the Conference Protocol specification. The 3rd (8th) column depicts the average time needed for calculating the discriminating sequence (if any).

The number of equivalent and different mutants (with respect to the used relation) are listed in the 4th and the 5th column of the Table 8.13 for the SIP Registrar specification and in the 9th column and in the 10th column for the Conference Protocol specification. Finally, the 6th and the 11th columns list the average time needed by the TGV tool to extract a final test case. Note that the third, the sixth, the eighth and the eleventh columns of the two tables list the average duration needed for the different steps during the test case generation. Thus, also the last row (Σ) lists the average values in these columns.

We have set the bound of the depth first search for the SIP Registrar to 5 steps and for the Conference Protocol to 10 steps.

Approximately 47% (31%) of the generated mutants for the SIP Registrar (Conference Protocol) specification were distinguishable from the original specification with respect to $\mathbf{ioco}^{|k|}$. The other mutants did not result in useful test cases when testing for particular faults. Although the chosen bound of the SIP Registrar was smaller than the bound used for the Conference Protocol, the conformance check on the SIP specification was slower. As for the minimization in Section 8.4.3, this was again because of the heavy branching of our SIP Registrar.

Table 8.12: Code coverage results.

Test Suite	SIP Registrar		Conference Protocol	
	Function Cov.	C/D Coverage	Function Cov.	C/D Coverage
Single	64%	26%	75%	58%
S	70%	31%	77%	60%
IL	67%	27%	77%	59%
OL	73%	34%	75%	60%
AL	73%	34%	75%	60%
T	73%	34%	77%	60%

Slicing-via-Test-Purpose

Table 8.14 lists for each mutation operator (1st column) the overall number of generated mutants (2nd and 6th column). In addition, Table 8.14 shows the average time needed for the particular test generation steps. The columns 3 and 7 list the time needed to extract a path that leads to the mutation using the CADP on-the-fly model-checker. Finally, this table depicts the average time needed by TGV to extract the relevant parts of the original specification S (4th and 8th column) and of the mutated specification S^M (5th and 9th column). Note that the final row (Σ) lists the average time values for all mutation operators. We instantiated the slicing test purpose of Figure 6.9 with $k = 1$.

As this table shows, the average time needed to generate the counterexample using the model-checker was 1 minutes and 49 seconds for the SIP Registrar specification and 18 minutes and 5 seconds for the Conference Protocol specification. As in the case of coverage based test purposes (see Section 8.6.1), the tools (e.g. TGV, model-checker) of the CADP toolbox ran out of memory on the SIP specification very fast. In contrast to that, for the Conference Protocol specification these tools ran for a long time until the memory expired.

Figure 8.6 serves to illustrate this effect. As this figure shows if the model-checker was able to extract a path leading to α then this was usually quite fast, i.e., for 75% of the SIP (Conference Protocol) mutants we needed at most 25 (4) seconds per mutant. However, if the model-checker failed to extract a path leading to α it took much more time on the Conference Protocol specification (failed on SIP: mean = seven minutes; failed on Conference Protocol: mean = one hour) before the model-checker ran out of memory. Note that in the case of the Conference Protocol we stopped the model-checker after one hour.

The two Tables 8.15 and 8.16 serve to illustrate the difference between using a bisimulation check as proposed by Aichernig and Corrales Delgado (2006) and using an ioco check as discussed in Section 6.4.4. These tables list for every mutation operator (1st column) the number of generated mutants (2nd and 9th column) and the number of mutants for which the model-checker ran out of memory (3rd and 10th column). For the SIP Registrar these tables also list the number of mutants on which TGV failed to extract the complete test graph (4th column). The counter-examples for these mutants led to states with no outgoing transitions (see Remark 6.2). Such a column is not needed for the Conference Protocol as there are no deadlock states in this specification. Furthermore, the two Tables 8.15 and 8.16 list the time needed for the bisimulation check and for the ioco check, respectively (5th and 11th column). The 6th (12th) column and the 7th (13th) column lists the number of equivalent (w.r.t. to the used relation) and non-equivalent mutants. Finally, the last column lists the time needed by TGV to derive the final test case.

As these two Tables show, the total number of generated test purposes, and thus the total number of generated test cases decreased by using the input-output conformance relation. Instead of using an over-approximated set of test cases comprising 317 test cases, we only needed to execute 279 test cases for the SIP specification. In the case of the Conference Protocol the ioco check led to 153 test cases instead of 193 test cases obtained by means of bisimulation. This was an improvement of approximately 20%.

Basically, the ioco-equivalent mutants come from mutations where input actions have been affected. For example, in our specification we use guards on inputs in order to avoid meaningless input actions. If

Table 8.13: Number of generated mutants and timing results for the extraction of the test cases for the two considered protocol specifications when using an bounded ioco-check.

oper-	SIP Registrar					Conference Protocol				
ator	mut.	ioco	=	$\neq$	tgv	mut.	ioco	=	$\neq$	tgv
ASO	11	4m52s	11	0	-	0	-	0	0	-
CRO	81	4m58s	26	55	5s	20	11s	12	8	1s
EDO	31	5m09s	14	17	5s	10	14s	7	3	1s
EIO	35	7m41s	24	11	5s	12	15s	9	3	1s
ENO	52	4m49s	16	36	5s	38	08s	21	17	1s
ERO	31	4m39s	7	24	5s	10	04s	4	6	1s
ESO	10	7m12s	8	2	5s	0	-	0	0	-
HDO	4	4m28s	1	3	5s	0	-	0	0	-
LRO	21	8m15s	13	8	5s	15	18s	15	0	-
MCO	33	8m12s	26	7	5s	30	18s	30	0	-
ORO	431	7m09s	245	186	5s	183	1m35s	126	57	1s
POR	6	32m39s	2	4	5s	3	07s	2	1	1s
PRO	18	5m03s	8	10	5s	30	09s	14	16	1s
RRO	91	7m42s	59	32	5s	0	-	0	0	-
SNO	32	6m57s	17	15	5s	39	18s	28	11	1s
SOR	0	-	0	0	-	0	-	0	0	-
USO	23	4m38s	5	18	5s	10	18s	8	2	1s
Σ	910	6m26s	482	428	5s	400	23s	276	124	1s

Table 8.14: Time needed for the different stages in the slicing via test purpose process.

oper-	SIP Registrar				Conference Protocol			
ator	mut.	MC	TGV S	TGV S^M	mut.	MC	TGV S	TGV S^M
ASO	11	6m47s	-	-	0	-	-	-
CRO	81	1m03s	35s	40s	20	6m57s	<1s	<1s
EDO	31	49s	34s	35s	10	4m43s	<1s	<1s
EIO	35	44s	32s	32s	12	3m56s	<1s	<1s
ENO	52	1m08s	35s	1m27s	38	5m49s	<1s	<1s
ERO	31	49s	34s	34s	10	4m41s	<1s	<1s
ESO	10	56s	36s	37s	0	-	-	-
HDO	4	33s	44s	44s	0	-	-	-
LRO	21	3m36s	37s	37s	15	1h42m39s	<1s	<1s
MCO	33	5m24s	35s	34s	30	1h43m24s	<1s	<1s
ORO	431	1m35s	36s	36s	183	19m23s	<1s	<1s
POR	-	-	-	-	-	-	-	-
PRO	-	-	-	-	-	-	-	-
RRO	91	3m15s	46s	1m04s	0	-	-	-
SNO	32	1m14s	40s	1m39s	39	15m55s	<1s	<1s
SOR	-	-	-	-	-	-	-	-
USO	23	1m01s	37s	38s	10	4m41s	<1s	<1s
Σ	886	1m49s	37s	44s	367	18m05s	<1s	<1s

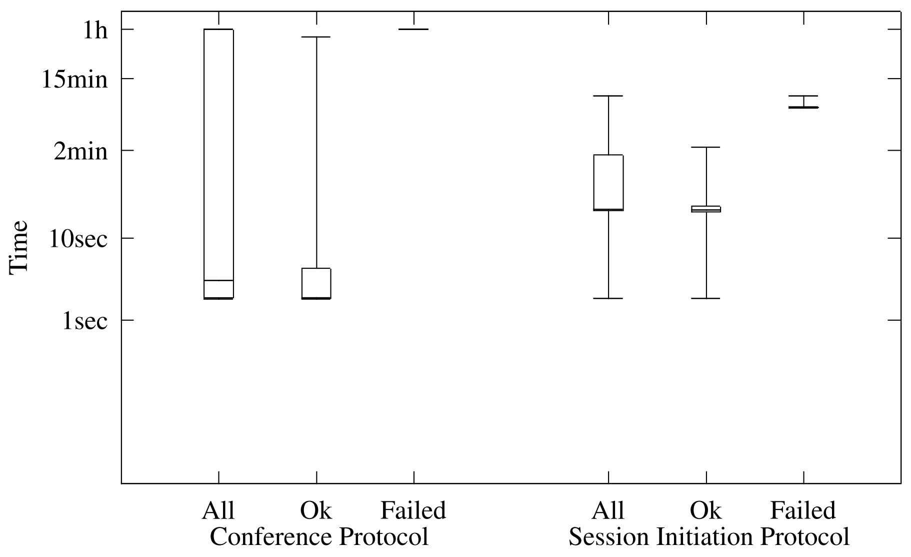


Figure 8.6: Minimum, maximum, 1^{st}, 2^{nd}, and 3^{rd} quartile of the counter-example generation time for the two different protocols (with a logarithmic scaled y-axis).

a mutation operator weakens such a condition the mutant allows more inputs than the original specification. However, when testing with respect to the input-output conformance relation injecting new additional inputs does not lead to failures, as shown in Section 6.4.2.

8.5.2 Test Case Execution Results

We executed the test cases obtained by mutation-based test purposes on the implementations of both protocols. Unfortunately, the commercial SIP Registrar was not available for these experiments. However, we applied all test cases to the OpenSER implementation.

Bounded Input-Output Conformance

Table 8.17 illustrates the number of passed (3rd and 7th column), failed (4th and 8th column) and inconclusive (5th and 9th column) verdicts obtained by executing the generated test cases. The number of test cases is listed in the 2nd column and the 6th column, respectively. Note that we ran the test cases of the SIP Registrar on two different configurations of the Kamailio implementation. For one test run authentication was turned on and for the other test run authentication was turned off. For example, we ran 15 test cases derived from the SNO mutations two times, resulting in 30 test runs. 24 out of these 30 test runs terminated with a pass verdict while 5 test runs reported a fail verdict. One of the 30 test runs led to an inconclusive verdict. A test case's verdict is inconclusive if the implementation chooses outputs different to the outputs required by the test case's preamble. That is, the chosen output is correct with respect to the specification, but the test case failed to bring the implementation to the required state.

For the Conference Protocol we have 27 faulty implementations. Thus, running for example the 8 test cases derived from the CRO mutations we got $8 \times 27 = 216$ test runs. 208 out of these 216 test runs terminated with a pass verdict, while 8 test runs ended with a fail verdict.

By the use of the generated test cases we detected 4 differences between the Kamailio Registrar and our specifications. However, a verdict fail does not imply that the corresponding mutant has been implemented. It also happened that a failure occurred during the execution of the preamble of the test case. The preamble

Table 8.15: Number of generated mutants and timing results for the extraction of the test cases for the two considered protocol specifications when using a bisimulation check.

oper-	SIP Registrar							Conference Protocol					
ator	mut.	∞	no δ	bisim	=	$\neq$	tgv	mut.	∞	bisim	=	$\neq$	tgv
ASO	11	11	-	-	-	-	-	0	-	-	-	-	-
CRO	81	8	9	1s	11	53	29s	20	0	1s	3	17	<1s
EDO	31	2	2	1s	13	14	29s	10	0	1s	0	10	<1s
EIO	35	2	3	1s	25	5	27s	12	0	1s	12	0	-
ENO	52	6	16	1s	4	26	28s	38	3	1s	8	27	<1s
ERO	31	2	2	1s	0	27	31s	10	0	1s	0	10	<1s
ESO	10	1	1	1s	7	1	33s	0	-	-	-	-	-
HDO	4	0	0	1s	2	2	36s	0	-	-	-	-	-
LRO	21	10	7	1s	2	2	24s	15	10	1s	1	4	<1s
MCO	33	25	5	1s	1	2	24s	30	21	1s	2	7	<1s
ORO	431	74	118	1s	96	143	25s	183	63	1s	35	85	<1s
POR	-	-	-	-	-	-	-	-	-	-	-	-	-
PRO	-	-	-	-	-	-	-	-	-	-	-	-	-
RRO	91	41	18	1s	13	19	26s	0	-	-	-	-	-
SNO	32	3	11	1s	8	10	11s	39	9	1s	7	23	<1s
SOR	-	-	-	-	-	-	-	-	-	-	-	-	-
USO	23	2	2	1s	6	13	29s	10	0	1s	0	10	<1s
Σ	886	187	194	1s	188	317	27s	367	106	1s	68	193	<1s

Table 8.16: Number of generated mutants and timing results for the extraction of the test cases for the two considered protocol specifications when using an ioco-check.

oper-	SIP Registrar							Conference Protocol					
ator	mut.	∞	no δ	ioco	=	$\neq$	tgv	mut.	∞	ioco	=	$\neq$	tgv
ASO	11	11	-	-	-	-	-	0	-	-	-	-	-
CRO	81	8	9	3s	35	29	18s	20	0	1s	11	9	<1s
EDO	31	2	2	7s	13	14	20s	10	0	<1s	7	3	<1s
EIO	35	2	3	1s	25	5	15s	12	0	-	12	0	-
ENO	52	6	16	<1s	6	24	19s	38	3	<1s	11	24	<1s
ERO	31	2	2	<1s	0	27	19s	10	0	<1s	3	7	<1s
ESO	10	1	1	1s	7	1	24s	0	-	-	-	-	-
HDO	4	0	0	<1s	2	2	24s	0	-	-	-	-	-
LRO	21	10	7	<1s	2	2	13s	15	10	<1s	1	4	<1s
MCO	33	25	5	<1s	1	2	13s	30	21	<1s	2	7	<1s
ORO	431	74	118	2s	107	132	19s	183	63	<1s	46	74	<1s
POR	-	-	-	-	-	-	-	-	-	-	-	-	-
PRO	-	-	-	-	-	-		-	-	-	-	-	-
RRO	91	41	18	<1s	15	17	22s	0	-	-	-	-	-
SNO	32	3	11	<1s	8	10	19s	39	9	<1s	8	22	<1s
SOR	-	-	-	-	-	-	-	-	-	-	-	-	-
USO	23	2	2	<1s	6	13	20s	10	0	<1s	7	3	<1s
Σ	886	187	194	2s	227	278	19s	367	106	<1s	108	153	<1s

Table 8.17: Test case execution results for the two protocol specifications using bounded input-output conformance checking.

oper-ator	SIP Registrar				Conference Protocol			
	no.tc.	pass	fail	inconc.	no.tc.	pass	fail	inconc.
ASO	0	-	-	-	0	-	-	-
CRO	55	85	25	0	8	208	8	0
EDO	17	27	7	0	3	78	3	0
EIO	11	18	4	0	3	78	3	0
ENO	36	60	12	0	17	430	29	0
ERO	24	37	11	0	6	156	6	0
ESO	2	2	2	0	0	-	-	-
HDO	3	5	1	0	0	-	-	-
LRO	8	15	1	0	0	-	-	-
MCO	7	13	1	0	0	-	-	-
ORO	186	302	70	0	57	1458	81	0
POR	4	7	1	0	1	26	1	0
PRO	10	17	3	0	16	408	24	0
RRO	32	56	8	0	0	-	-	-
SNO	15	24	5	1	11	282	15	0
SOR	0	-	-	-	0	-	-	-
USO	18	29	7	0	2	52	2	0
Σ	428	698	158	1	124	3176	172	0

is the sequence of messages that aims to bring the implementation to a certain state in which the difference between the mutant and the original specification can be observed.

For the Conference Protocol we detected in total 7 of the 27 faulty implementations. Recall that the bound of the test case length is ten actions in the case of the Conference Protocol. This limitation in the length of the test cases is mainly the reason why we did not detect more faulty implementations.

Slicing-via-Test-Purpose

The Tables 8.18 and 8.19 illustrate the number of test cases (2nd and 6th column), the number of passed (columns 3 and 7), failed (columns 4 and 8) and inconclusive (columns 5 and 9) test runs when using the bisimulation relation (Table 8.18) and when using the ioco relation (Table 8.19) for every mutation operator (1st column). By the use of the generated test cases we detected three different failures in the OpenSER Registrar and ten faulty implementations of the Conference Protocol.

As illustrated by the Tables 8.18 and 8.19 the test cases obtained by means of bisimulation led to more fail verdicts. However, a verdict fail does not imply that the corresponding mutant has been implemented. It also happened that a failure occurred during the execution of the preamble of the test case. The preamble is the sequence of messages that aims to bring the implementation to a certain state in which the difference between the mutant and the original specification can be observed. Furthermore, a test case's verdict is inconclusive if the implementation chooses outputs different to the outputs required by the test case's preamble. That is, the chosen output is correct with respect to the specification, but the test case failed to bring the implementation to the required state.

8.5.3 Discussion

Mutation Operators Remarkably the two mutation operators ASO and SOR did not lead to any mutant with an observable difference for the two specifications. Within the Conference Protocol there were no

Table 8.18: Test case execution results for the two protocol specifications using the slicing via test purpose approach (bisimulation check).

oper-	OpenSER Registrar				Conference Protocol			
ator	no.tc.	pass	fail	inconc.	no.tc.	pass	fail	inconc.
ASO	-	-	-	-	-	-	-	-
CRO	53	25	4	77	17	408	23	28
EDO	14	5	0	23	10	172	10	88
EIO	5	2	0	8	0	-	-	-
ENO	26	13	1	38	27	339	34	356
ERO	27	18	0	36	10	171	10	89
ESO	1	0	0	2	-	-	-	-
HDO	2	2	0	2	-	-	-	-
LRO	2	1	0	3	4	104	4	0
MCO	2	1	0	3	7	114	4	71
ORO	143	65	15	206	85	1146	96	1053
POR	-	-	-	-	-	-	-	-
PRO	-	-	-	-	-	-	-	-
RRO	19	14	0	24	-	-	-	-
SNO	10	8	0	12	23	531	28	62
SOR	-	-	-	-	-	-	-	-
USO	13	4	0	22	10	171	10	89
Σ	317	158	20	456	193	3156	219	1836

logical expressions comprising more than one logical operator. Thus the ASO operator did not produce any mutant on that specification. On the contrary, there were logical expressions within the SIP Registrar specification that use more than one logical operator. However, these expressions were always conjunctions of terms, thus an association shift within these expression did not lead to any observable difference. As there were no enabling (») nor any disabling (`[>`) statements within the Conference Protocol specifications the SOR operator does not produce any mutant. The SIP Registrar specification comprised six » statements. However, in this specification replacing » with `[>` always resulted in an invalid specification. This was because for the `[>` operator the exit-behavior, i.e. the types of the returned values, had to be equal. Unfortunately, the exit-behavior always differed in our specification. Anyway, on other specifications these mutation operators can lead to mutants exhibiting a **ioco** incorrect behavior.

Bisimulation vs. Input-Output Conformance For the slicing via test purpose approach we were able to compare the results using the bisimulation check (Aichernig and Corrales Delgado, 2006) with the results of using an ioco check; the available bisimulation checker required the construction of the state spaces in advance. The tests generated on basis of the ioco relation detected the same failures as the test cases generated on basis of the bisimulation relation. In our case the additional test cases of the bisimulation approach did not reveal additional failures. Consequently, these additional test cases fail because of failures unrelated to their mutants. As the main goal of mutation testing is to check if a particular mutant has been implemented, using the ioco relation for ioco testing leads to the exact set of non-equivalent mutants.

Slicing-via-Test-Purpose vs. Bounded IOCO In the case of the SIP Registrar specification the bounded ioco approach revealed more different mutants than the slicing-via-test-purpose technique. This is basically, because we used one action (plus the length of the preamble) as upper bound for the latter approach. However, as indicated by the results for the Conference Protocol our slicing-via-test-purpose approach potentially detects more different mutants than the bounded conformance check.

Table 8.19: Test case execution results for the two protocol specifications using the slicing via test purpose approach (ioco check).

oper-	OpenSER Registrar				Conference Protocol			
ator	no.tc.	pass	fail	inconc.	no.tc.	pass	fail	inconc.
ASO	-	-	-	-	-	-	-	-
CRO	29	35	1	22	9	224	19	0
EDO	14	15	0	13	3	77	4	0
EIO	5	7	0	3	-	-	-	-
ENO	24	25	1	22	24	513	27	108
ERO	27	31	0	23	7	179	10	0
ESO	1	1	0	1	-	-	-	-
HDO	2	2	0	2	-	-	-	-
LRO	2	3	0	1	4	108	0	0
MCO	2	3	0	1	7	174	15	0
ORO	132	128	2	134	74	906	39	1053
POR	-	-	-	-	-	-	-	-
PRO	-	-	-	-	-	-	-	-
RRO	17	16	0	18	-	-	-	-
SNO	10	6	0	14	22	293	4	297
SOR	-	-	-	-	-	-	-	-
USO	13	15	0	11	3	77	4	0
Σ	278	287	4	265	153	2551	122	1458

8.6 Coverage-based Test Purposes

We also derived test suites based on the coverage criteria presented in Section 6.5 for both protocols. This section reports on the obtained results when using coverage-based test purposes on our two applications under test.

8.6.1 Test Case Generation Results

We have implemented a tool that takes a LOTOS specification, automatically generates probe annotated specifications for the coverage criteria presented in Section 6.5.1, and instantiates our generic test purposes for the inserted probes.

For the SIP Registrar specification our tool generated 10 probe annotated specifications for process coverage (P), 27 probe annotated specifications for action coverage (A), and 39 probe annotated specifications for condition coverage (C), decision coverage (D), condition/decision (CD) coverage and modified condition/decision coverage (MCDC), respectively. For decision, condition, condition/decision, and for modified condition/decision coverage the number of generated test purposes does not only depend on the number of inserted probes but also on the number of conditions within a decision. For those coverage criteria our tool generated 78 (D), 98 (C), 176 (CD), and 94 (MCDC) test purposes, respectively.

For the Conference Protocol specification our tool derived 16 probe annotated specifications for process coverage, 26 probe annotated specifications for action coverage and 32 probe annotated specifications for each of the logical coverage criteria. The tool generated 64 (D), 78 (C), 142 (CD), and 71 (MCDC) test purposes, respectively.

These generated test purposes together with the annotated specifications served as a basis for our experimental evaluation.

Table 8.20: Test case generation results for the Session Initiation Protocol.

C.	No.	Regular				Reduced				
	TP	ok	∞	time	Cov.	ok	cov	∞	time	Cov.
P	10	10	0	09m	100.0	2	8	0	08m	100.0
A	27	25	2	2h49m	92.6	10	15	2	2h37m	92.6
D	78	72	6	8h28m	92.3	10	62	6	3h11m	92.3
C	98	94	4	11h13m	95.9	12	82	4	3h26m	95.9
CD	176	166	10	19h39m	94.3	12	154	10	4h30m	94.3
MCDC	94	55	39	10h26m	58.5	32	26	36	9h44m	61.7
Σ	483	422	61	2d04h44m		78	347	58	23h36m	

Generating Test Cases Based on Coverage Criteria Only

The Tables 8.20 and 8.21 list the results when generating test cases for the coverage criterion based test purposes (1st column). These tables show the number of instantiated test purposes (2nd column), the number of generated test cases (3rd and 7th column), the number of test purposes where TGV failed to generate a test case (4th and 9th column) and the time needed to process all test purposes (5th and 10th column).

TGV failed on some test purposes, because its depth first search algorithm ran into a path not leading to an accept state within the synchronous product between the specification and the test purpose. As our specifications had infinite state spaces and our test purposes lack *Refuse* states TGV eventually ran out of memory. Furthermore, we may generate test purposes for which no test case can be derived, e.g. if there are conditions/decisions that are tautologies or condition/decisions that are unsatisfiable. Also for MCDC we may generate test purposes for which no test case exists (see Section 6.5.5). However, we did not observe invalid test purposes in our experiments.

The left part of these tables depict the figures obtained when generating test cases for all test purposes. In contrast, the right part illustrates the results gained from reducing the test suites as described in Section 6.5.7. An additional column (8th column) in this part lists the number of probes additionally covered by the generated test cases.

TGV ran out of memory on 61 (73) test purposes for the SIP (Conference Protocol) application (see 4th column). Thus, for the Session Initiation Protocol we got test suites having 100% process coverage, 92.6% action coverage, 92.3% decision coverage, 95.9% condition coverage, 94.3% CD coverage, and 58.5% MCDC coverage. For the Conference Protocol the generated test suites had 93.8% process coverage, 73.1% action coverage, 87.5% decision coverage, 84.6% condition coverage, 85.9% CD coverage, and 64.8% MCDC coverage, respectively.

When generating test cases for uncovered test purposes only (see Section 6.5.7) we were sometimes able to cover probes for which TGV runs out of memory otherwise. This is because TGV may run out of memory before the branch containing the probe can be selected during the depth first search. For example, for the SIP application the modified condition/decision coverage increased from 58.5% to 61.7%.

However, as guaranteed by our minimization approach there is never a reduction of model coverage, with respect to the coverage criterion, although the overall number of test cases had been reduced by 82% and by 38% for the SIP application and the Conference Protocol Application, respectively.

The test case generation takes much longer for the Conference Protocol than for the Session Initiation Protocol. Figure 8.7 serves to illustrate this effect. As this figure shows if TGV is able to generate a test case for a particular test purpose this is usually quite fast, i.e., on average it takes approximately seven minutes for the SIP application and approximately six seconds for the Conference Protocol application. However, if TGV fails to generate a test case for a particular test purpose it takes a long period of time (SIP: on average

Table 8.21: Test case generation results for the Conference Protocol.

C.	No.	Regular				Reduced				
	TP	ok	∞	time	Cov.	ok	cov	∞	time	Cov.
P	16	15	1	6h56m	93.8	3	12	1	6h56m	93.8
A	26	19	7	2d03h16m	73.1	12	7	7	2d03h13m	73.1
D	64	56	8	4d17h05m	87.5	37	20	7	1d18h55m	89.1
C	78	66	12	5d21h38m	84.6	46	21	11	2d11h29m	85.9
CD	142	122	20	5d11h33m	85.9	78	46	18	4d06h20m	87.3
MCDC	71	46	25	7d16h40m	64.8	25	22	24	5d07h20m	66.2
Σ	397	324	73	26d05h08m		201	128	68	16d06h13m	

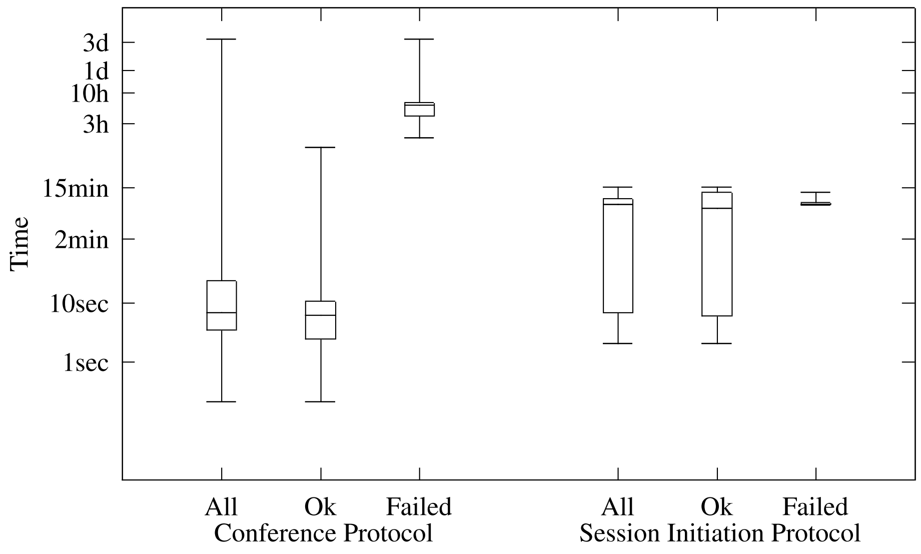


Figure 8.7: Minimum, maximum, 1^{st}, 2^{nd}, and 3^{rd} quartile of the test case generation times for the two different protocols (with a logarithmic scaled y-axis).

eight minutes, Conference Protocol: on average six hours) before it runs out of memory. In particular, for the Conference Protocol it sometimes takes days before TGV runs out of memory.

Supplementing Manually Designed Test Purposes

This section comprises the results when combining manually designed test purposes with coverage based test purposes (see Section 6.5.6). For this experiments we used the test purposes of Section 8.4, i.e. five test purposes for Session Initiation Protocol and ten test purposes for the Conference Protocol. TGV derived one test case for every test purpose.

Table 8.22 contains information about the test suites generated when combining coverage based test purposes with hand-crafted test purposes. It shows for each coverage criterion (1st column) the number of test cases (2nd column and 7th column) contained in the initial test suite, i.e., the test suite directly derived from the hand-crafted test purposes. The 3rd and the 8th column list the coverage value for these initial test suites. The number of missed probes is given by the sum of the successive two columns (i.e., 4th and 5th; 9th and 10th). In these columns the table comprises the number of coverage based test purposes for which TGV succeeded to generate test cases (4th and 9th column) and the number of test purposes for which TGV

Table 8.22: Complementing manually generated test cases.

Cov. Crite-rion	Session Initiation Protocol					Conference Protocol				
	No TC	Cov-erage	No.TC ok	No.TC ∞	New Cov.	No TC	Cov-erage	No.TC ok	No.TC ∞	New Cov.
P	5	90.0%	1	0	100.0%	10	100.0%	0	0	100.0%
A	5	66.7%	8	1	96.3%	10	96.2%	0	1	96.2%
D	5	59.0%	28	4	94.9%	10	90.6%	6	0	100.0%
C	5	52.0%	45	2	98.0%	10	74.4%	19	1	98.7%
CD	5	31.8%	112	8	95.5%	10	50.7%	62	8	94.4%
MCDC	5	30.9%	26	39	58.5%	10	74.7%	3	15	78.9%

Table 8.23: Test execution results using the regular test suite on two different SIP Registrars.

C.	OpenSER				commercial			
	pass	fail	inconc.	failures	pass	fail	inconc.	failures
P	18	1	1	1	18	1	1	1
A	40	1	9	1	27	15	8	2
D	100	16	28	3	72	44	28	3
C	124	25	39	3	86	65	37	3
CD	224	41	67	3	158	109	65	3
MCDC	45	38	27	3	46	37	27	3
Σ	551	122	171	3	407	271	166	3

ran out of memory (5th and 10th column). Finally, in the columns six and eleven the table lists the new coverage values for the extended test suites.

By complementing the test purpose using the presented technique we observed an increase of coverage for both specifications. On average we were able to increase the coverage for the Session Initiation Protocol by 36% and by 14% for the Conference Protocol.

8.6.2 Test Case Execution Results

Table 8.23 shows the results obtained when executing the test cases generated for a certain coverage criterion (1st column) against our two SIP Registrar implementations in terms of the issued verdicts passed (2nd and 6th column), failed (3rd and 7th column), and inconclusive (4th and 8th column) for the OpenSER Registrar (2nd-5th column), and the commercial Registrar (6th-9th column). Furthermore, this table shows the number of detected differences between the implementations and the specification (5th and 9th column). Note that for the fifth column and for the ninth column the last row (Σ) shows the total number of failures detected by the union of all test suites. This is also the case for all failures labeled columns in all successive tables.

We executed our test cases against the implementations using two different configurations covered by our specification (see Section 8.2). The values in Table 8.23 are the sums of the values for runs with the two configurations.

The 122 failed test cases on the OpenSER Registrar implementation detected three different failures, and the 271 failed test cases for the commercial implementation also revealed three different failures. On the contrary, our five manually designed test purposes (see Table 8.26) detected one and two failures in the OpenSER and in the commercial Registrar implementation. In that case the automatically generated test cases using our coverage based approach outperformed the manually designed test purposes.

Table 8.24: Test execution results using the reduced test suite on two different SIP Registrars.

C.	OpenSER				commercial			
	pass	fail	inconc.	failures	pass	fail	inconc.	failures
P	2	1	1	1	2	1	1	1
A	12	0	8	0	3	10	7	1
D	10	2	8	2	2	10	8	2
C	12	2	10	2	2	12	10	2
CD	11	2	11	2	1	13	10	2
MCDC	2	36	26	3	3	35	26	3
Σ	49	43	64	3	13	81	62	3

Table 8.24 illustrates the results when executing the reduced test suites on the two SIP Registrar implementations. Even though our reduction of the test suite size does not affect the coverage value, it is known that the fault sensitivity can be adversely affected. This can be observed in Table 8.24, where the reduction of the number of test cases within the test suites also reduced the number of detected failures. The test suites based on condition/decision coverage, on decision coverage and on condition coverage missed one failure that had been revealed by their non-reduced counterparts. The action coverage test suite did not detect any failures in the open source implementation and missed two failures in the commercial implementation.

Table 8.25 lists the result when executing the generated test cases on the 27 faulty implementations of the Conference Protocol. This table comprises the results for the regular (2nd to 5th column) and for the reduced (6th to 9th column) test suites. As this table shows, our coverage based test cases detected 8 of the 27 faulty implementations. We missed failures mainly because we fail to generate test cases for some test purposes. Furthermore, the generated test cases capture similar scenarios of the specification. For example, we never observed the generation of a test purposes for the scenario of users leaving a conference. Thus, we missed all failures that require at least one leave action of a conference user. On the contrary, our manually designed test purposes used leave messages and detected five failures (missed by coverage based testing) which occurred only because of the use of leave messages.

As for the SIP test suites, in some cases the test suite reduction also reduced the fault sensitivity for the Conference Protocol test suites.

While the execution of some test cases on the SIP Registrars led to inconclusive verdicts there were no inconclusive verdicts when testing the Conference Protocol implementations. Due to the structure of the specifications the test cases for Conference Protocol did not comprise any inconclusive verdict state. This is, because TGV generates an inconclusive verdict if either a refuse state or a sink state was reached during test case generation. As we did not have refuse states in our test purposes and as there were no sink states in the Conference Protocol specifications there were no inconclusive verdicts in the derived test cases.

On the contrary, the SIP Registrar specification had sink states because we had an upper bound for the message sequence number in our specification. Thus, there were sink states when this upper bound was reached, which led to inconclusive verdicts during test case generation.

Results for Complemented Test Suites

The tables 8.26 and 8.27 show the results obtained when running the complemented test suites on the SIP applications and on the Conference Protocol applications, respectively.

Table 8.26 shows the number of test cases (2nd column) used for the different test suites. Furthermore, this table depicts the number of passed (3rd and 7th column), the number of failed (4th and 8th column), and the number of inconclusive (5th and 9th column) verdicts obtained from the test runs. Finally, the sixth and the tenth column list the number of failures detected in the different implementations.

Table 8.25: Test execution results using the regular and the reduced test suites on the 27 faulty implementations of the Conference Protocol.

C.	Regular				Reduced			
	pass	fail	inconc.	failures	pass	fail	inconc.	failures
P	403	2	0	1	80	1	0	1
A	497	16	0	4	309	15	0	4
D	1347	165	0	8	952	47	0	7
C	1609	173	0	8	1116	126	0	7
CD	2956	338	0	8	1886	220	0	8
MCDC	1134	108	0	8	581	94	0	8
Σ	7946	802	0	8	4924	503	0	8

Table 8.26: Test execution results using the complemented test suites on two different SIP Registrars.

C.	No. TC	OpenSER				commercial			
		pass	fail	inconc.	failures	pass	fail	inconc.	failures
manual TPs	5	5	1	4	1	3	2	5	2
P	6	6	1	5	1	3	3	6	2
A	13	15	1	10	1	5	10	11	3
D	33	23	7	36	2	3	32	31	3
C	50	41	14	45	3	11	51	38	4
CD	117	130	25	79	3	75	93	66	4
MCDC	31	18	10	34	3	3	31	28	4
Σ	255	238	59	213	3	103	222	185	4

We again ran each test suite on the two different configurations of our SIP Registrars, i.e., each test case runs on two different configurations. Therefore, we have twice as many verdicts as test cases. The first line of the table shows the results of only running the test cases derived from the manually created test purposes.

Using the complemented test suites we found two failures for each implementation not detected when using test cases derived from hand-crafted test purposes only. However, also the test suite sizes increased. While having ten test runs (five test cases) for our hand-crafted test purposes we have at most 234 test runs (117 test cases) for CD coverage.

Table 8.27 uses the structure of Table 8.26 and lists the results obtained when executing the complemented test suite for the Conference Protocol Specification on the 27 faulty implementations.

For the Conference Protocol we detected one more faulty implementation. The test suite size increased from ten test cases for hand-crafted test purposes to at most 72 test cases when using CD coverage for complementation. In the case of condition coverage the test suite size had been tripled while the number of detected failures had been increased by 5%.

8.6.3 Discussion

Satisfiability of Coverage Items Our current approach does not consider whether the coverage items are feasible within the specification. For example, we may have a test purpose that aims to cover a condition's false outcome but the condition is a tautology. Especially, for modified condition/decision coverage it may happen that we use truth-value assignments which can never occur within our specification. However, if a specification's state space is of reasonable size, then TGV will terminate without finding a test case.

Table 8.27: Test execution results using the complemented test suite on the Conference Protocol implementations.

C.	No.TC	pass	fail	inconc.	failures
manual TPs	10	195	75	0	17
P	10	195	75	0	17
A	10	195	75	0	17
D	16	328	104	0	17
C	29	650	133	0	18
CD	72	1678	266	0	18
MCDC	13	262	89	0	17
Σ	160	3503	817	0	18

Refuse States for Test Purposes On some test purposes TGV ran out of memory for the used specifications. As already mentioned, this was basically because TGV's depth first search algorithm, the infinity of our specification's state spaces and because our test purposes lack *Refuse* states. One idea for adding refuse states is to calculate the relevant behavior by means of the data and control flow graph of a LOTOS specification. By inserting additional markers one can refuse irrelevant behavior.

Weak/Strong Coverage Except for action coverage our test purposes only guarantee weak coverage (see Section 6.5.3). An interesting question is whether there is a difference in the failure detection ability of a test suite generated with respect to strong coverage.

Complementing Manual Test Purposes An important insight is that the coverage of the test cases derived from hand-crafted test purposes is not satisfactory. Although we tried to derive the test purposes by identifying relevant scenarios from the informal specifications we missed some parts of the model. For example, the test purposes designed for the Session Initiation Protocol failed to cover a whole process. Although complementing manual test purposes significantly improved the specification coverage the number of detected mutants of the Conference Protocol had only been increased by 5%. This shows that a high specification coverage does not guarantee in general a high failure detection rate on real implementations. However, our complemented test suites always detected more failures than the test suites derived from manual test purposes only.

8.7 Comparison

In this section we relate the results presented in this chapter to each other. Furthermore, we compare our results on the Conference Protocol with other approaches that have been applied to this protocol specification.

8.7.1 Comparison of the Presented Techniques

In addition to applying our approaches we used random testing, i.e. the TORX tool (Tretmans and Brinksma, 2003), on our specifications. For this random testing we ran the TORX tool 5000 times on OpenSER SIP Registrar and 100 times on every Conference Protocol implementation. For our comparison we conducted each experiment three times with different seeds for the random value generator, e.g. for the Conference Protocol we made 100 random test runs and repeated this experiment three times. The results shown in this thesis are the average values out of these three experiments. Thus, for the OpenSER SIP Registrar we got

Table 8.28: Overview of the main results using random, test-purpose-based, fault-based, and coverage-based test case generation techniques.

IUT	Technique	seq. length	test gen.	test cases	avg. coverage F	avg. coverage C/D	no. failures
SIP	Random	10.95	4s	5000	73%	38%	4
	Manual TP (1)	2.20	1s	5	64%	26%	1
	Manual TP (many)	4.53	1s	6813	73%	34%	6
	Bounded ioco	3.75	6m26s	428	70%	31%	4
	Slicing-via-tp	3.94	3m31s	279	72%	35%	3
	Coverage (reduced)	3.91	18m09s	78	73%	36%	3
	Coverage (compl.)	3.24	1m23s	255	73%	36%	3
Conf. Prot.	Random	9.03	4s	100	70%	56%	19
	Manual TP (1)	5.53	1s	10	75%	58%	17
	Manual TP (many)	4.98	1s	408	77%	60%	23
	Bounded ioco	7.23	23s	124	66%	51%	7
	Slicing-via-tp	8.21	18m06s	153	74%	57%	10
	Coverage (reduced)	30.77	1h57m	201	66%	50%	8
	Coverage (compl.)	15.46	1h05m	160	69%	57%	18

5000 test runs (i.e. the average of 3 times 5000), while for the Conference Protocol we have 2700 test runs, i.e. 100 test runs for each of the 27 faulty implementations.

Due to the underlying assumptions of the **ioco** relation TORX may obtain incorrect verdicts on the Conference Protocol specification (see Section 5.6). Therefore, the Conference Protocol specification includes models of buffers that formalize the asynchronous communication between the implementation and its environment. Because of these buffers TORX is applicable for testing the Conference Protocol even in an asynchronous environment.

Table 8.28 summarizes the results when testing the two protocols using different test case selection strategies. This table shows for each of the three techniques (2nd column), the average length of the executed test sequences (3rd column). The next columns depict the average time needed to generate a single test case (4th column) and the overall number of generated test cases (5th column). In addition, Table 8.28 shows the code coverage[‡] in terms of function coverage (6th column), and condition/decision coverage (7th column). Finally, Table 8.28 illustrates the number of detected failures (8th column). Note that we list the results obtained when testing the OpenSER SIP Registrar in this table.

This comparison of the different testing techniques indicates that the quality of test suites derived from test purposes highly depends on the tester's skills. Using one test case per test purpose revealed only one failure in the SIP Registrar. On the contrary, this technique led to a test suite detecting 17 faulty Conference Protocol implementations. Using multiple test cases per test purpose did not only improve the number of detected failures but also the code coverage on the real implementations. For both specifications random testing outperformed test purpose based testing with one test case per test purpose. However, using multiple test cases per test purpose revealed more failures than random testing.

In the case of the SIP Registrar, the condition/decision (C/D) coverage achieved by random testing was higher than the C/D coverage from test purpose based testing (multiple test cases per test purpose). That means that random testing had inspected the covered functions of the SIP Registrar more thoroughly. On the other hand, also the coverage based test suites achieved a higher C/D coverage than the test purpose based test suites. However, as different parts of the code were exercised the number of detected failures differed.

[‡]For coverage measurements we used the Bullseye Coverage Tool: http://www.bullseye.com

In the case of the SIP Registrar the fault-based test cases (bounded ioco, slicing-via-tp) and the coverage based test cases covered more source code and found more failures than one test case per scenario. For the Conference Protocol this was not the case. This is another evidence that scenario based testing highly depends on the skills of the tester.

While random testing, limited to at most 25 steps, produced the longest test runs (on average) for the SIP Registrar, the coverage based test cases were longer than the random test runs for the Conference Protocol. This is because of TGV's depth first search algorithm and because of the structure of the Conference Protocol specification. For the SIP Registrar there was not much difference in the average test run length for the test cases from manual test purposes, from fault-based testing, and from coverage based testing. In contrast to that, fault-based testing led to slightly longer test runs than manual test purposes on the Conference Protocol.

The average time needed to derive a single test case was much higher for coverage based test purposes and for slicing-via-test-purpose than for the other techniques. This is basically because we failed to generate test cases for some test purposes. If we failed, it took a long time until the tools ran out of memory (see Section 8.5.1 and Section 8.6.1).

For the Conference Protocol test cases generated based on scenarios detected all failures that had been detected by the other approaches. That is, combining all test cases of all our experiments we detected 23 of the 27 faulty implementations. Using many test cases for a single scenario revealed failures that have not been detected using random, fault-based or coverage-based testing. The same was true for the OpenSER SIP Registrar.

8.7.2 Comparison with Existing Techniques

The Conference Protocol case study (Terpstra et al., 1996) used in this thesis was designed to support the comparison of different test case selection strategies. Consequently, others have applied various testing techniques to this protocol specification. Table 8.29 gives an overview of the results achieved by different testing techniques.

By the use of *random* test case selection 25 of the 27 erroneous implementations of the Conference Protocol have been found (Belinfante et al., 1999). The two missed mutants accept data units from any source, i.e. they do not check if the data units come from potential conference partners. As this behavior is not captured by the specification, these two mutants are correct with respect to the specification. Also the probabilistic TORX as presented by Feijs et al. (2000) had been applied to the Conference Protocol (Goga, 2003): also 25 mutants had been detected.

Du Bousquet et al. (2000) applied the TGV tool for testing the Conference Protocol. By the use of 19 *manually designed test purposes*, 24 faulty implementations have been found. However, even after ten hours of test purposes design they did not manage to generate a test suite that detected the 25th mutant. Because they used the same specification as Belinfante et al. (1999), two of the mutants were correct with respect to the used specification. Unfortunately, the test purposes used by du Bousquet et al. (2000) were not available to us. Thus, we developed our own set of test purposes. Our test purposes led to a test suite which detects 17 mutants (2nd last row of Table 8.29).

The AUTOLINK tool (Schmitt et al., 1998) was able to reveal 22 of the 27 faulty implementations (Goga, 2001). As AUTOLINK relies on test purposes the quality of the generated test suite is dependent on the skills of the tester. However, from a theoretical point of view, the conformance relation employed by AUTOLINK is weaker than **ioco** (Goga, 2001). Thus, AUTOLINK has less failure detection power.

Heerink et al. (2000) applied the Philips Automated Conformance Tester (PHACT) to the Conference Protocol. As PHACT relies on extended finite state machines (EFSM) they developed an EFSM specification for the Conference Protocol. By the use of this specification PHACT was able to detect 21 of the 27 faulty implementations.

Table 8.29: Comparison of test results among different test case selection strategies using the Conference Protocol.

selection strategy/tool	no.failures	reference
TORX (Random)	25	Belinfante et al. (1999)
Probabilistic TORX (Random)	25	Goga (2003)
Coverage on LTS[§]	25	Huo and Petrenko (2009)
TGV (Manual TP (1))	24	du Bousquet et al. (2000)
AUTOLINK	22	Goga (2001)
PHACT	21	Heerink et al. (2000)
SPECEXPLORER[¶]	3	Botinčan and Novaković (2007)
Manual TP (1)	17	Table 8.10 in Section 8.4.2
Manual TP (many)	23	Table 8.10 in Section 8.4.2
Bounded ioco	7	Table 8.17 in Section 8.5.2
Slicing-via-tp	10	Table 8.19 in Section 8.5.2
Coverage (reduced)	8	Table 8.25 in Section 8.6.2
Coverage (complemented)	18	Table 8.27 in Section 8.6.2

Microsoft's SPECEXPLORER tool (Campbell et al., 2005) had been applied to the Conference Protocol by (Botinčan and Novaković, 2007). Unfortunately, they did not use the 27 available faulty implementations but developed 3 mutants by their own. Thus, the obtained results can neither be compared with the other case studies nor with our results.

Coverage based test case selection on the level of the labeled transition system Huo and Petrenko (2009) led to test suites that also detected 25 of the 27 faulty mutants. However, Huo and Petrenko (2009) did not rely on the LTS derived from the LOTOS specification but manually created an LTS showing the behavior of the Conference Protocol.

Although random test case generation outperforms almost all other approaches, this does not render them useless. The efficiency of the random walk has been known for a long time (West, 1989). However, first of all, a single case study is not sufficient for a general statement about the performance of different testing techniques. Second, the 27 mutants of the Conference Protocol are "functional" mutants with high-level fault models (Belinfante et al., 1999), i.e. no output (mutants do not send outputs where they are supposed to), no internal checks (mutants do not check their internal data structures), no internal update (mutants do not update their internal data structures). These restricted fault models could be the reason for the observed performance of the different techniques.

Finally, a clear drawback of random testing is that there is no inherent stopping criteria. While the other approaches have a fixed sized test suite, random testing can continue forever. Thus in practice random testing needs to be complemented with coverage measurements or with upper bounds in order to eventually stop testing.

8.8 Test Case Grouping Results

In this section we report on the results obtained when applying our approach for grouping test case grouping (see Chapter 7). As the erroneous implementations of the Conference Protocol only comprise one failure per implementation we have used the Session Initiation Protocol for the evaluation of our approach.

[§]Unfortunately, Botinčan and Novaković (2007) used a different set of mutants and thus their results are not directly comparable with the other approaches.

[¶]Huo and Petrenko (2009) used their own LTS and, thus, did not rely on one of the available specifications.

Table 8.30: Test suite details for the Session Initiation Protocol Registrar.

Test Suite	Impl.	No. Test Cases	Verdicts Pass	Verdicts Fail	Failures
Test Purposes (Aichernig et al., 2007)	Comm.	6644	1700	4944	9
	OpenSER	6644	4587	2057	4
Random (Weiglhofer and Wotawa, 2008b)	Comm.	500	207	293	3
	OpenSER	500	237	263	5

Table 8.31: Failure report for the commercial Session Initiation Protocol Registrar.

Test Suite	No. Failures	Failed TCs	Failure Identifier									
			1	2	3	4	5	6	7	8	9	10
Test Purpose	9	4944	360	2012	214	174	130	246	436	389	983	-
Random	3	293	-	212	-	-	-	-	-	75	-	6

For assessing the quality of our test case grouping approach we used two test suites: (1) The first test suite has been obtained by heuristically selecting test cases from the complete test graph generated by TGV; (2) We used the random test suites obtained by applying TORX (see Section 8.7.1). Both test suites have been applied to the commercial Session Initiation Protocol Registrar and to the OpenSER implementation.

Executing the different test suites on the two implementations led to the results shown in Table 8.30. This table shows the number of test cases (3rd column) of a particular test suite (1st column) that have been executed on one of the two Registrar implementations (2nd column). Furthermore, this table shows the number of obtained pass (4th column) and fail (5th column) verdicts and the total number of detected failures (6th column).

The two tables 8.31 and 8.32 show the number of failed test cases for the different detected failures. As there can be seen, there are some failures that caused many test cases to fail. For example, in the case of test purpose based testing the OpenSER implementation failure 4 causes 1131 test cases to fail.

8.8.1 Grouping Characteristics

An optimal test case grouping has one group per detected failure and all groups are homogeneous, i.e., all test cases within a group failed because of the same discrepancy between the implementation and its specification. We use some characteristic figures for quantifying the quality of our automated grouping. Note that all characteristics rely on the knowledge of the detected failures. Thus we manually analyzed and identified the detected failures for every test run. In the following we use $f(tc_i)$ to denote the real failure detected by test case tc_i.

We use the following characteristics:

- Number of Groups: The number of groups that have been generated by grouping the failed test cases.

Table 8.32: Failure report for the OpenSER Session Initiation Protocol Registrar.

Test Suite	No. Failures	Failed TCs	Failure Identifier				
			1	2	3	4	5
Test Purpose	4	2057	295	344	287	1131	-
Random	5	263	48	50	28	13	124

- Test Cases per Group (No.TC): The number of test cases contained in a group.
- Inhomogeneity (IH): If one takes one test case from a group, the inhomogeneity says on average how many test cases failed because of a different failure. The inhomogeneity is a measure for the regularity of the groups. It gives insight into the group structure by counting the differences between the failures detected by the test cases of one group $g = \{tc_1, \dots, tc_n\}$:

  $$IH(g) = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} d(tc_i, tc_j)}{\sum_{i=1}^{n-1} i}$$

  where

  $$d(tc_i, tc_j) = \begin{cases} 1 & \text{if } f(tc_i) = f(tc_j) \\ 0 & \text{otherwise} \end{cases}$$

  If all test cases of a group g detected the same failure then $IH(g) = 0$. On the contrary, if every test case revealed a different failure then we have $IH(g) = 1$.
- Failure Count Difference (FCD): The difference between the number of groups and the number of detected failures.
- Single Failure Groups (SFG): How many groups have an inhomogeneity of 0, i.e., how many groups only comprise test cases which all have failed because of the same failure.
- Reported Failures (RF): The number of different failures that one can report if only one test case per group is considered. Basically, this number is given by the count of the covered failures when using only the groups with an inhomogeneity of 0. Note that the number of reported failures is at least one.

8.8.2 Test Case Grouping Results and Evaluation

The Table 8.33 lists the characteristics of the groups obtained in our experiments. This table shows for each test suite (1st column), for each implementation (2nd column), and for each partitioning of LOTOS specifications (3rd column; P = process, A = action, D = decision, C = condition) the number of failures of the implementation (4th column). Furthermore, this table comprise the number of generated groups (5th column) and the number of test cases (6th-8th column) in terms of the minimum number (6th column), the average number (7th column), and the maximum number (8th column) over all groups. In addition, Table 8.33 comprises information about the inhomogeneity (minimum, maximum, and average over all groups). Finally, this table lists the failure count difference (FCD; 12th column), the number of single failure groups (SFG; 13th column), the number of reported failures (RF; 14th column), and the percentage of reported failures (%RF; 15th column), i.e. how many failures of the existing failures can be identified.

Note that the number of reported failures is even one if there is no group with an inhomogeneity of zero (second row of Table 8.33). This is because we analyze one test case per group. Thus, at least one failed test case is analyzed and we can report on one failure.

As Table 8.33 shows partitioning the SIP specification with respect to its processes was too coarse: The difference between the number of failures within the implementations and the number of generated groups (FCD) were even negative in one case. Also the inhomogeneity of these groups was relatively high, i.e. from 0.23 to 0.45. For example 0.45 means that when one picks a test case from a group on average 45% of the other test cases of this group failed because of a different failure. On the contrary, in the best case the average inhomogeneity over all groups was 4%.

The partitioning with respect to the conditions of the SIP Registrar specification worked best. The number of failures that could be reported when analyzing one test case per group was equal to the number of the detected failures of our implementations. However, in the case of using conditions the number of generated groups was always higher than the number of failures detected by our test cases.

In terms of test cases to be analyzed the improvements gained by our approach can be summarized as follows: For the test purpose based test suite one had to look at 39 (OpenSER) and at 43 (Commercial

Table 8.33: Test case grouping results for the SIP Registrar using a similarity threshold of 1.0.

Test Suite	Impl.	Blocks	No. Fail.	No. Grp.	No. TC			Inhomogeneity			FCD	SFG	RF	%RF
					min.	avg.	max.	min.	avg.	max.				
Test Pur- poses	OpenSER	P	4	5	12	411.40	572	0.00	0.41	0.75	1	1	1	25.0
	Commercial	P	9	5	36	988.80	2144	0.01	0.45	0.74	-4	0	1	11.1
	OpenSER	A	4	17	1	121.00	414	0.00	0.05	0.67	13	13	4	100.0
	Commercial	A	9	18	2	274.67	1916	0.00	0.26	0.83	9	7	5	55.6
	OpenSER	D	4	25	1	82.28	414	0.00	0.05	0.67	21	20	4	100.0
	Commercial	D	9	26	2	190.15	1916	0.00	0.25	0.83	17	10	6	66.7
	OpenSER	C	4	39	1	52.74	414	0.00	0.04	0.67	35	34	4	100.0
	Commercial	C	9	43	1	114.98	1916	0.00	0.20	0.83	35	21	9	100.0
Ran- dom	OpenSER	P	5	5	3	52.60	144	0.00	0.23	0.68	0	2	2	40.0
	Commercial	P	3	5	5	58.60	130	0.00	0.39	0.60	2	1	1	33.3
	OpenSER	A	5	27	1	9.74	48	0.00	0.27	1.00	22	14	5	100.0
	Commercial	A	3	21	1	13.95	130	0.00	0.29	1.00	18	10	3	100.0
	OpenSER	D	5	27	1	9.74	68	0.00	0.22	1.00	22	17	5	100.0
	Commercial	D	3	24	1	12.21	130	0.00	0.32	1.00	21	11	3	100.0
	OpenSER	C	5	47	1	5.60	68	0.00	0.18	1.00	42	32	5	100.0
	Commercial	C	3	35	1	8.37	130	0.00	0.29	1.00	32	18	3	100.0

Implementation) test cases. On the contrary, without grouping one needed to analyze 2057 and 4944 test cases, respectively. For the SIP Registrars it took us approximately 15 seconds to analyze a single failed test case. Thus, analyzing the test results after grouping took approximately 10 minutes for each implementation. On the contrary, without grouping we needed more than 8 hours to identify the failures detected in the OpenSER implementation and more than 20 hours for the failures of the commercial implementation.

For the random test suites the improvements were 88% (approximately nine minutes instead of one hour and 13 minutes) for the commercial and 82% (approximately 12 minutes instead of one hour) for the OpenSER implementation.

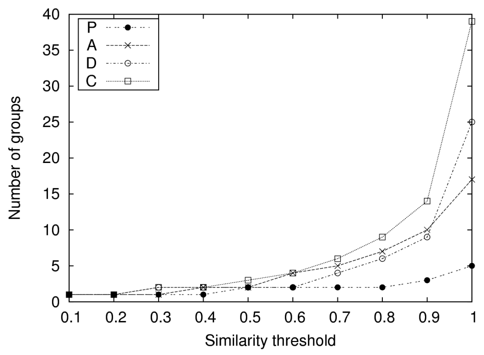
8.8.3 Impact of the Threshold Value

Our classification relies on a threshold value. This value is used to decide on the similarity between two test cases. The main factors for the quality of the generated groups are this threshold and the chosen partitioning of the specification. The three Figures 8.8a, 8.8b, and 8.9 illustrate this effect.

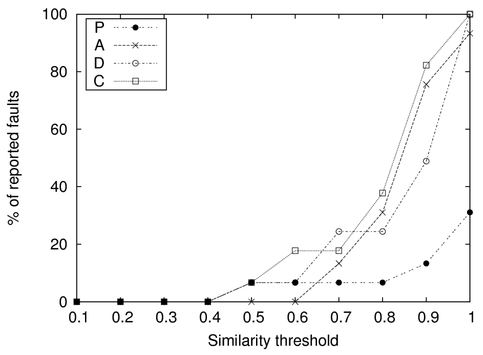
Figure 8.8a shows for each partitioning method of Section 7.3, the number of groups above the similarity threshold. This graph uses the data obtained when classifying the test purpose based test runs of the OpenSER Registrar. As the graphs for our other test runs look similar we do not show them here. As one can see from this figure, the number of generated groups increased when increasing the similarity threshold. The increase in the number of groups was more significant for the finer partitioning criteria, i.e. partitioning with respect to conditions (C) led to more groups than a partitioning with respect to processes.

Figure 8.8b illustrates the effect of the similarity threshold on the number of reported failures. Note that we do not use the absolute number of grouped failures in this figure, but the percentage value. 100% means that for every failure there exists at least one group such that the test cases within that group solely fail because of one failure. For this figure, we use the average over all our experiments. As Figure 8.8b shows, threshold values below 0.8 were not meaningful. In the best case (conditions) for a threshold of 0.8 we identified approximately 40% of the detected failures by using one test case per group. However, when using a similarity threshold value of 1.0 and conditions or decisions as partitioning criteria, then in the case of the SIP Registrars one can report all detected failures.

Figure 8.9 shows the average inhomogeneity of all groups over the threshold value. Although the average inhomogeneity does not decrease monotonically, higher threshold values usually showed a smaller inhomogeneity.



(a) Correlation of the number of groups and the similarity threshold.



(b) Change of the number of reported failures with respect to the similarity threshold.

Figure 8.8: Impact of the similarity threshold on the group results for the different partitioning methods for LOTOS specifications.

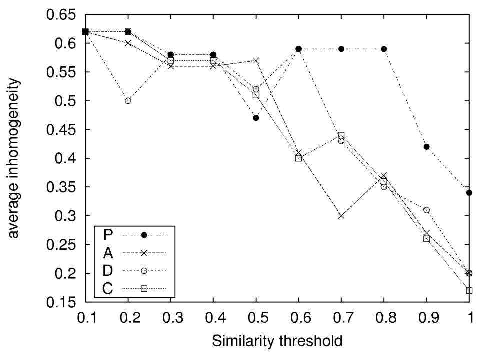


Figure 8.9: Change of the average inhomogeneity with respect to the similarity threshold for the different partitioning methods for LOTOS specifications.

Although the number of groups was closer to the number of failures when using a small threshold value (see Figure 8.8a), the quality of the groups increased when increasing the threshold value (Figure 8.8b). For example, when using conditions as partitioning criterion and a threshold value of 0.9 we get homogeneous groups for 80% of the failures. The average inhomogeneity is 0.26.

Finally, Figure 8.10 shows the results when comparing the groups generated by our approach with randomly selecting failed test cases. Therefore, we randomly drew n different test cases from the set of failed test cases, where n denotes the number of groups. We repeated this experiment 1000 times for every n with $1 \leq n \leq 50$. As this figure illustrates, except for process coverage, our grouping approach outperformed the random selection of test cases.

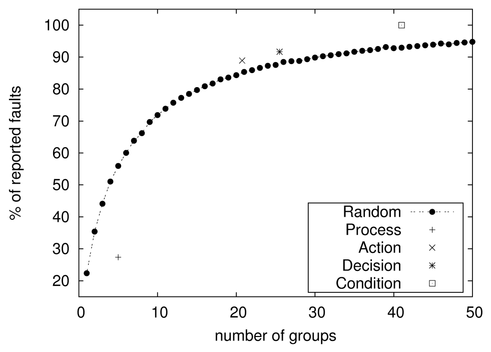


Figure 8.10: Random grouping versus using spectrum based fault localization for test case grouping.

Chapter 9

Conclusion

This chapter summarizes the findings and results of the previous chapters and gives an outlook to further research.

9.1 Summary

This thesis started with a brief overview of software testing, putting the topic in its proper context. Subsequently, the input-output conformance (ioco) testing theory had been summarized, and extensions and modifications of this theory had been surveyed. After that this thesis presented a reformulation of the ioco theory in terms of the Unified Theories of Programming and showed how to assist test engineers in developing formal test objectives and analyzing test case execution results.

The different parts of this thesis show the broad range of research options in conformance testing, and software testing in general. Starting from a rather theoretical work on a denotational semantics for ioco to the generating of formal test objectives, and the analysis of test case execution results. Although the techniques presented in this thesis make input-output conformance testing more amenable in practice there are still many challenges that need to be addressed.

Applying model-based testing techniques almost always leads to the question: Where does the model come from? While research usually assumes that the model is just there, creating a model is a challenging task. The model needs to be as simple as possible but still concrete enough to provide enough information for deriving useful test cases. The effort needed to developed the model is one of the main barriers, preventing a wide use of model-based testing techniques.

However, even if one decides to use model-based testing techniques and commits to the development of a model another issue is the correctness of the model. Model-based testing techniques generally assume that the model is correct. As software development in general, the construction of a model is a creative and complex task. Although a model is usually more abstract, and thus simpler than a real piece of software, it is still hard to get the model right. Thus, model validation is an important issue and software engineers need to be supported not only in the creation but also in the validation of their models.

Regardless of all these challenges that arise when applying model-based testing techniques we believe that the time spent on modelling the software system pays off in terms of software quality. Not only that the testing process becomes systematic and focused but also the system's behavior is expressed precisely and unambiguously. Also the modeling task enforces software engineers to rethink the requirements which are always beneficial thoughts.

9.2 Further Research

Albeit this thesis tackles some of issues of model-based conformance testing, there are plenty of challenges that need further investigation.

Unified Testing Theories While this thesis recasts the input-output conformance theory in the Unifying Theories of Programming (UTP) framework, it would be interesting to study other testing theories by means of UTP, like for example, mutation testing of sequential programs which has already been explored in the UTP framework (Aichernig and He, 2009). Formalizing the different theories in terms of UTP would ease the comparison of the different approaches and give a unified picture of testing. Having formal representations of the assumptions of different approaches simplifies the selection of a proper technique for a particular testing problem. Furthermore, we believe that having all theories embedded in a single framework helps when teaching students, because students do not need to learn new formalisms for understanding the various concepts and theories.

Unified Test Case Generation The denotational semantics of the input-output conformance theory presented in this thesis facilitate the use of SAT/SMT solvers (e.g. (Dutertre and de Moura, 2008)) for test case generation. The model together with its healthiness conditions and the healthiness conditions of test cases can be reformulated as a satisfiability problem. Test case selection techniques can be integrated. For example, mutation testing on reactive processes would require that a mutant $\mathcal{P}^m$ does not conform to the original process $\mathcal{P}$, i.e. $\mathcal{P} \not\sqsubseteq_{ioco} \mathcal{P}^m$. If other testing theories get embedded in the UTP framework, then existing test case selection strategies can be simply combined with these theories. Furthermore, the implementation of a test case generator is then reduced to the substitution of the healthiness conditions for the satisfiability problem.

Refuse States for Test Purposes Chapter 6 presents techniques to improve test purpose based testing. While the experimental results showed the applicability of these techniques to industrially sized case studies there is still room for improvements. Test purposes become more efficient if they make use of refuse states. The more refuse states are used the fewer parts of the model need to be considered for test case generation. For example, the approach for coverage-based test purposes does not use any refuse state. One may use the data flow and the control flow of a specification to calculate the relevant behavior of a specification. Experiments on such an idea have been presented in the paper (Weiglhofer and Wotawa, 2009b). Another way to reduce the model's size is to use program slicing (Weiser, 1981; Tip, 1995) with respect to the test objective on the level of the model. This may also speed up test case generation.

Symbolic and Timed Test Purposes In this thesis we focused on the generation of test purposes for labeled transition systems. However, test purpose based test case generation has also been applied to symbolic (Rusu et al., 2000; Jeannet et al., 2005; Gaston et al., 2006) and timed (David et al., 2008b,a) transition systems. An extension of the approaches presented in this thesis would be necessary to deal with the additional challenges introduced by an explicit notion of data (symbolic) and time.

Test Suite Reduction In practice there is always insufficient time for testing and test suite reduction is a widely studied area of research (e.g. (Rothermel et al., 2002)). It is inefficient to first generate test suites and later reduce the number of test cases of the test suites. Due to the use of a formal conformance relation we have a precise notion of failures. We can use this knowledge in order to skip irrelevant coverage items or particular mutations. For example for the ioco relation we can skip mutations when we know that an additional input is introduced. Such a mutant cannot be distinguished from the original specification with respect to ioco. Also coverage criteria can be optimized in that way, as only the outputs are relevant.

Improving Test Result Analysis Our approach for grouping failed test cases with respect to the detected failures currently only relies on model coverage. If one has access to the covered parts of the source code, then this information can be incorporated into our grouping approach. Having both, information of the model coverage and information of the source code coverage may improve the result of test case grouping.

Appendix A

Proofs

A.1 Chapter 4 (Unifying Input-Output Conformance)

A.1.1 Section 4.2 (IOCO Specifications)

Lemma 4.1 (IOCO1-idempotent).

$$\mathbf{IOCO1} \circ \mathbf{IOCO1} = \mathbf{IOCO1}$$

Proof of Lemma 4.1.

$\mathbf{IOCO1}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (ok \Rightarrow (wait' \vee ok'))$ (prop. calculus)
$= P \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(P)$

Lemma 4.2 ($\mathbb{I}$-IOCO1-healthy).

$$\mathbf{IOCO1}(\mathbb{I}) = \mathbb{I}$$

Proof of Lemma 4.2.

$\mathbf{IOCO1}(\mathbb{I}) =$ (def. of **R3**)
$= (\neg ok \wedge (tr \leq tr') \vee ok' \wedge \ldots) \wedge (ok \Rightarrow (wait' \vee ok'))$ (prop. calculus)
$= (\neg ok \wedge (tr \leq tr') \wedge (\neg ok \vee wait' \vee ok')) \vee (ok' \wedge \cdots \wedge (\neg ok \vee wait' \vee ok'))$ (prop. calculus)
$= (\neg ok \wedge (tr \leq tr') \wedge \neg ok) \vee (\neg ok \wedge (tr \leq tr') \wedge wait') \vee (\neg ok \wedge (tr \leq tr') \wedge ok') \vee$
$\quad (ok' \wedge \cdots \wedge \neg ok) \vee (ok' \wedge \cdots \wedge wait') \vee (ok' \wedge \cdots \wedge ok')$ (prop. calculus)
$= (\neg ok \wedge (tr \leq tr')) \vee (ok' \wedge \ldots)$ (prop. calculus)
$= \mathbb{I}$

Lemma 4.3 (commutativity-IOCO1-R1).

$$\mathbf{IOCO1} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{IOCO1}$$

Proof of Lemma 4.3.

$\mathbf{IOCO1}(\mathbf{R1}(P)) =$ (def. of **R1**)
$= \mathbf{IOCO1}(P \wedge (tr \leq tr'))$ (def. of **IOCO1**)
$= P \wedge (tr \leq tr') \wedge (ok \Rightarrow (wait' \vee ok'))$ (prop. calculus)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (tr \leq tr')$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(P) \wedge (tr \leq tr')$ (def. of **R1**)
$= \mathbf{R1}(\mathbf{IOCO1}(P))$

Lemma 4.4 (commutativity-IOCO1-R2).

$$\mathbf{IOCO1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{IOCO1}$$

Proof of Lemma 4.4.

$\mathbf{IOCO1}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)
$= \mathbf{IOCO1}(P(\langle\rangle, tr' - tr))$ (def. of **IOCO1**)
$= P(\langle\rangle, tr' - tr) \wedge (ok \Rightarrow (wait' \vee ok'))$ (tr’,tr are not used in **IOCO1**)
$= (P \wedge (ok \Rightarrow (wait' \vee ok')))(\langle\rangle, tr' - tr)$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{IOCO1}(P))$

Lemma 4.5 (commutativity-IOCO1-R3).

$$\mathbf{IOCO1} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{IOCO1}$$

Proof of Lemma 4.5.

$\mathbf{IOCO1}(\mathbf{R3}(P)) =$ (def. of **R3**)
$= \mathbf{IOCO1}(\mathbb{I} \lhd wait \rhd P)$ (def. of **IOCO1**)
$= (\mathbb{I} \lhd wait \rhd P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (distr. of ∧ over if)
$= (\mathbb{I} \wedge (ok \Rightarrow (wait' \vee ok'))) \lhd wait \rhd (P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbb{I}) \lhd wait \rhd \mathbf{IOCO1}(P)$ (Lemma 4.2)
$= \mathbb{I} \lhd wait \rhd \mathbf{IOCO1}(P)$ (def. of **R3**)
$= \mathbf{R3}(\mathbf{IOCO1}(P))$

Lemma 4.6 (closure-;-IOCO1).

$$\mathbf{IOCO1}(P;Q) = P;Q \text{ provided P and Q are } \mathbf{IOCO1} \text{ and } \mathbf{R3} \text{ healthy}$$

Proof of Lemma 4.6.

$$\mathbf{IOCO1}(P;Q) = \qquad \text{(assumption)}$$

$$= \mathbf{IOCO1}(\mathbf{IOCO1}(P);\mathbf{IOCO1}(\mathbf{R3}(Q))) \qquad \text{(def. of } \mathbf{R3}\text{)}$$

$$= \mathbf{IOCO1}(\mathbf{IOCO1}(P);\mathbf{IOCO1}(\mathbb{I} \lhd wait \rhd Q)) \qquad \text{(def. of ;)}$$

$$= \mathbf{IOCO1}(\exists v_0 \bullet \mathbf{IOCO1}(P)[v_0/v'] \wedge \mathbf{IOCO1}(\mathbb{I} \lhd wait \rhd Q)[v_0/v]) \qquad \text{(def. of } \mathbf{IOCO1}\text{, if)}$$

$$\begin{aligned} = \mathbf{IOCO1}(\exists v_0 \bullet (P \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v'] \wedge \\ ((\mathbb{I} \wedge wait \vee \neg wait \wedge Q) \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v]) \end{aligned} \qquad \text{(prop. calculus)}$$

$$= \mathbf{IOCO1}\left(\exists v_0 \bullet \left((P \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v'] \wedge \begin{pmatrix} (\mathbb{I} \wedge wait \wedge (ok \Rightarrow (wait' \vee ok')))\vee \\ (\neg wait \wedge Q \wedge (ok \Rightarrow (wait' \vee ok'))) \end{pmatrix} [v_0/v]\right)\right)$$

$$\text{(def. of [])}$$

$$= \mathbf{IOCO1}\left(\exists v_0 \bullet \left((P[v_0/v'] \wedge (ok \Rightarrow (wait_0 \vee ok_0))) \wedge \begin{pmatrix} (\mathbb{I}[v_0/v] \wedge wait_0 \wedge (ok_0 \Rightarrow (wait' \vee ok')))\vee \\ (\neg wait_0 \wedge Q[v_0/v] \wedge (ok_0 \Rightarrow (wait' \vee ok'))) \end{pmatrix}\right)\right)$$

$$\text{(prop. calculus)}$$

$$= \mathbf{IOCO1}\left(\exists v_0 \bullet \begin{pmatrix} (P[v_0/v'] \wedge (ok \Rightarrow (wait_0 \vee ok_0)) \wedge \mathbb{I}[v_0/v] \wedge wait_0 \wedge (ok_0 \Rightarrow (wait' \vee ok')))\vee \\ (P[v_0/v'] \wedge (ok \Rightarrow (wait_0 \vee ok_0)) \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (ok_0 \Rightarrow (wait' \vee ok'))) \end{pmatrix}\right)$$

$$\text{(def. of } \mathbf{IOCO1}\text{)}$$

$$= \exists v_0 \bullet \left(\begin{pmatrix} P[v_0/v'] \wedge \mathbb{I}[v_0/v] \wedge wait_0 \wedge \\ (ok \Rightarrow (wait_0 \vee ok_0))\wedge \\ (ok_0 \Rightarrow (wait' \vee ok'))\wedge \\ (ok \Rightarrow (wait' \vee ok')) \end{pmatrix} \vee \begin{pmatrix} P[v_0/v'] \wedge Q[v_0/v] \wedge \neg wait_0 \wedge \\ (ok \Rightarrow (wait_0 \vee ok_0))\wedge \\ (ok_0 \Rightarrow (wait' \vee ok'))\wedge \\ (ok \Rightarrow (wait' \vee ok')) \end{pmatrix}\right) \qquad \text{(prop. calculus)}$$

$$= \exists v_0 \bullet \left(\begin{pmatrix} P[v_0/v'] \wedge \mathbb{I}[v_0/v] \wedge wait_0 \wedge \\ (ok \Rightarrow (wait_0 \vee ok_0))\wedge \\ (ok_0 \Rightarrow (wait' \vee ok'))\wedge \\ (ok \Rightarrow (wait' \vee ok')) \end{pmatrix} \vee \begin{pmatrix} P[v_0/v'] \wedge Q[v_0/v] \wedge \neg wait_0 \wedge \\ (ok \Rightarrow (wait_0 \vee ok_0))\wedge \\ (ok_0 \Rightarrow (wait' \vee ok')) \end{pmatrix}\right) \qquad \text{(prop. calculus)}$$

$$= \exists v_0 \bullet \left(\begin{pmatrix} P[v_0/v'] \wedge \mathbb{I}[v_0/v] \wedge wait_0 \wedge \\ (ok \Rightarrow (wait_0 \vee ok_0))\wedge \\ (ok_0 \Rightarrow (wait' \vee ok')) \end{pmatrix} \vee \begin{pmatrix} P[v_0/v'] \wedge Q[v_0/v] \wedge \neg wait_0 \wedge \\ (ok \Rightarrow (wait_0 \vee ok_0))\wedge \\ (ok_0 \Rightarrow (wait' \vee ok')) \end{pmatrix}\right) \qquad \text{(def. of [])}$$

$$= \exists v_0 \bullet \left(\begin{pmatrix} P \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v'] \wedge \\ (\mathbb{I} \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v] \wedge wait_0 \end{pmatrix} \vee \begin{pmatrix} (P \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v'] \wedge \\ (Q \wedge (ok \Rightarrow (wait' \vee ok')))[v_0/v] \wedge \neg wait_0 \end{pmatrix}\right)$$

$$\text{(def. } \mathbf{IOCO1}\text{)}$$

$$= \exists v_0 \bullet \big((\mathbf{IOCO1}(P)[v_0/v'] \wedge (\mathbf{IOCO1}(\mathbb{I}))[v_0/v] \wedge wait_0) \vee (\mathbf{IOCO1}(P)[v_0/v'] \wedge (\mathbf{IOCO1}(Q))[v_0/v] \wedge \neg wait_0)\big)$$

$$\text{(assumption)}$$

$$= \exists v_0 \bullet \big((P[v_0/v'] \wedge (\mathbf{IOCO1}(\mathbb{I}))[v_0/v] \wedge wait_0) \vee (P[v_0/v'] \wedge Q[v_0/v] \wedge \neg wait_0)\big) \qquad \text{(Lemma 4.2)}$$

$$= \exists v_0 \bullet \big((P[v_0/v'] \wedge \mathbb{I}[v_0/v] \wedge wait_0) \vee (P[v_0/v'] \wedge Q[v_0/v] \wedge \neg wait_0)\big) \qquad \text{(Lemma 4.2)}$$

$$= \exists v_0 \bullet \big((P[v_0/v'] \wedge \mathbb{I}[v_0/v] \wedge wait_0) \vee (P[v_0/v'] \wedge Q[v_0/v] \wedge \neg wait_0)\big) \qquad \text{(prop. calculus)}$$

$$= \exists v_0 \bullet P[v_0/v'] \wedge ((\mathbb{I}[v_0/v] \wedge wait_0) \vee (Q[v_0/v] \wedge \neg wait_0)) \qquad \text{(def. of if)}$$

$$= \exists v_0 \bullet P[v_0/v'] \wedge (\mathbb{I}[v_0/v] \lhd wait_0 \rhd Q[v_0/v]) \qquad \text{(def. of [])}$$

$$= \exists v_0 \bullet P[v_0/v'] \wedge (\mathbb{I} \lhd wait \rhd Q)[v_0/v] \qquad \text{(assumption)}$$

$$= \exists v_0 \bullet P[v_0/v'] \wedge Q[v_0/v] \qquad \text{(def. of ;)}$$

$$= P;Q$$

Lemma 4.7 (IOCO2-idempotent).

$$\mathbf{IOCO2} \circ \mathbf{IOCO2} = \mathbf{IOCO2}$$

Proof of Lemma 4.7.

$\mathbf{IOCO2}(\mathbf{IOCO2}(P)) =$ (def. of **IOCO2**)

$= \mathbf{IOCO2}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of **IOCO2**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of **IOCO2**)

$= \mathbf{IOCO2}(P)$

Lemma 4.8 (IOCO3-idempotent).

$$\mathbf{IOCO3} \circ \mathbf{IOCO3} = \mathbf{IOCO3}$$

Proof of Lemma 4.8.

$\mathbf{IOCO3}(\mathbf{IOCO3}(P)) =$ (def. of **IOCO3**)

$= \mathbf{IOCO3}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*)))$ (def. of **IOCO3**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*)))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*)))$ (def. of **IOCO3**)

$= \mathbf{IOCO3}(P)$

Lemma 4.9 (commutativity-IOCO2-IOCO3).

$$\mathbf{IOCO2} \circ \mathbf{IOCO3} = \mathbf{IOCO3} \circ \mathbf{IOCO2}$$

Proof of Lemma 4.9.

$\mathbf{IOCO2}(\mathbf{IOCO3}(P)) =$ (def. of **IOCO3**)

$= \mathbf{IOCO2}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*))))$ (def. of **IOCO2**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*)))$ (def. of **IOCO2**)

$= \mathbf{IOCO2}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\widehat{\ }\delta^*)))$ (def. of **IOCO3**)

$= \mathbf{IOCO3}(\mathbf{IOCO2}(P))$

Lemma 4.10 (commutativity-IOCO2-R1).

$$\mathbf{IOCO2} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{IOCO2}$$

Proof of Lemma 4.10.

$\mathbf{IOCO2}(\mathbf{R1}(P)) =$	(def. of **R1**)
$= \mathbf{IOCO2}(P \wedge (tr \leq tr'))$	(def. of **IOCO2**)
$= P \wedge (tr \leq tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$	(prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \wedge (tr \leq tr')$	(def. of **IOCO2**)
$= \mathbf{IOCO2}(P) \wedge (tr \leq tr')$	(def. of **R1**)
$= \mathbf{R1}(\mathbf{IOCO2}(P))$	

Lemma 4.11 (commutativity-IOCO2-R2).

$$\mathbf{IOCO2} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{IOCO2}$$

Proof of Lemma 4.11.

$\mathbf{IOCO2}(\mathbf{R2}(P(tr, tr'))) =$	(def. of **R2**)
$= \mathbf{IOCO2}(P(\langle\rangle, tr' - tr))$	(def. of **IOCO2**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$	(tr',tr are not used in **IOCO2**)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))))(\langle\rangle, tr' - tr)$	(def. of **IOCO2**)
$= \mathbf{IOCO2}(P)(\langle\rangle, tr' - tr)$	(def. of **R2**)
$= \mathbf{R2}(\mathbf{IOCO2}(P))$	

Lemma 4.12 (commutativity-IOCO2-IOCO1).

$$\mathbf{IOCO2} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{IOCO2}$$

Proof of Lemma 4.12.

$\mathbf{IOCO2}(\mathbf{IOCO1}(P)) =$	(def. of **IOCO1**)
$= \mathbf{IOCO2}(P \wedge (ok \Rightarrow (wait' \vee ok')))$	(def. of **IOCO2**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$	(prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \wedge (ok \Rightarrow (wait' \vee ok'))$	(def. of **IOCO2**)
$= \mathbf{IOCO2}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$	(def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{IOCO2}(P))$	

Lemma 4.13 (commutativity-IOCO3-R1).

$$\mathbf{IOCO3} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{IOCO3}$$

Proof of Lemma 4.13.

$\mathbf{IOCO3}(\mathbf{R1}(P)) =$ (def. of **R1**)
$= \mathbf{IOCO3}(P \wedge (tr \le tr'))$ (def. of **IOCO3**)
$= P \wedge (tr \le tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \widehat{\ } \delta^*)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \widehat{\ } \delta^*))) \wedge (tr \le tr')$ (def. of **IOCO3**)
$= \mathbf{IOCO3}(P) \wedge (tr \le tr')$ (def. of **R1**)
$= \mathbf{R1}(\mathbf{IOCO3}(P))$

Lemma 4.14 (commutativity-IOCO3-R2).

$$\mathbf{IOCO3} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{IOCO3}$$

Proof of Lemma 4.14.

$\mathbf{IOCO3}(\mathbf{R2}(P(tr,tr'))) =$ (def. of **R2**)
$= \mathbf{IOCO3}(P(\langle\rangle, tr' - tr))$ (def. of **IOCO3**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \widehat{\ } \delta^*)))$ (substitute tr',tr)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet (tr' - tr) - \langle\rangle \in s \widehat{\ } \delta^*))))(\langle\rangle, tr' - tr)$ (def. of **IOCO3**)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \widehat{\ } \delta^*))))(\langle\rangle, tr' - tr)$ (def. of **IOCO3**)
$= \mathbf{IOCO3}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{IOCO3}(P))$

Lemma 4.15 (commutativity-IOCO3-IOCO1).

$$\mathbf{IOCO3} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{IOCO3}$$

Proof of Lemma 4.15.

$\mathbf{IOCO3}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{IOCO3}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **IOCO3**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \widehat{\ } \delta^*)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \widehat{\ } \delta^*))) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO3**)
$= \mathbf{IOCO3}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{IOCO3}(P))$

Lemma 4.16 (commutativity-IOCO3-IOCO2).

$$\mathbf{IOCO3} \circ \mathbf{IOCO2} = \mathbf{IOCO2} \circ \mathbf{IOCO3}$$

Proof of Lemma 4.16.

$\mathbf{IOCO3}(\mathbf{IOCO2}(P)) =$ (def. of **IOCO2**)

$= \mathbf{IOCO3}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))))$ (def. of **IOCO3**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \wedge$

$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*)))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*))) \wedge$

$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of **IOCO3**)

$= \mathbf{IOCO3}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of **IOCO2**)

$= \mathbf{IOCO2}(\mathbf{IOCO3}(P))$

Lemma 4.17 (commutativity-IOCO2-R3$^\delta$).

$$\mathbf{IOCO2} \circ \mathbf{R3}^\delta = \mathbf{R3}^\delta \circ \mathbf{IOCO2}$$

Proof of Lemma 4.17.

$\mathbf{IOCO2}(\mathbf{R3}^\delta(P)) =$ (def. of $\mathbf{R3}^\delta$)

$= \mathbf{IOCO2}(\mathbb{I}^\delta \lhd wait \rhd P)$ (def. of **IOCO2**)

$= (\mathbb{I}^\delta \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ ($\wedge$-if-distr)

$= (\mathbb{I}^\delta \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))) \lhd wait \rhd$

$\quad (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))))$ (def. of if and $\neg wait$)

$= \mathbb{I}^\delta \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))))$ (def. of **IOCO2**)

$= \mathbb{I}^\delta \lhd wait \rhd (\mathbf{IOCO2}(P))$ (def. of $\mathbf{R3}^\delta$)

$= \mathbf{R3}^\delta(\mathbf{IOCO2}(P))$

Lemma 4.18 (commutativity-IOCO3-R3$^\delta$).

$$\mathbf{IOCO3} \circ \mathbf{R3}^\delta = \mathbf{R3}^\delta \circ \mathbf{IOCO3}$$

Proof of Lemma 4.18.

$\mathbf{IOCO3}(\mathbf{R3}^\delta(P)) =$ (def. of $\mathbf{R3}^\delta$)

$= \mathbf{IOCO3}(\mathbb{I}^\delta \lhd wait \rhd P)$ (def. of **IOCO3**)

$= (\mathbb{I}^\delta \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*)))$ ($\wedge$-if-distr)

$= (\mathbb{I}^\delta \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*)))) \lhd wait \rhd$

$\quad (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*))))$ (def. of if and $\neg wait$)

$= \mathbb{I}^\delta \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*))))$ (def. of **IOCO3**)

$= \mathbb{I}^\delta \lhd wait \rhd (\mathbf{IOCO3}(P))$ (def. of $\mathbf{R3}^\delta$)

$= \mathbf{R3}^\delta(\mathbf{IOCO3}(P))$

Lemma 4.19 (closure-;-IOCO2).

$$\mathbf{IOCO2}(P;Q) = P;Q \text{ provided P and Q are } \mathbf{IOCO2} \text{ and } \mathbf{R3}^{\delta} \text{ healthy}$$

Proof of Lemma 4.19.

$\mathbf{IOCO2}(P;Q) =$ (assumption and def. of $\mathbf{R3}^{\delta}$ and ;)

$= \mathbf{IOCO2}(\exists v_0 \bullet P[v_0/v'] \wedge (\mathbb{I}^{\delta} \lhd wait \rhd Q)[v_0/v])$ (def. $\mathbf{IOCO2}$, if, and substitution)

$= \exists v_0 \bullet P[v_0/v'] \wedge (\mathbb{I}^{\delta}[v_0/v] \wedge wait_0 \vee \neg wait_0 \wedge Q[v_0/v]) \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence))$

(def. of $\mathbb{I}^{\delta}$ and prop. calculus)

$$= \exists v_0 \bullet \left(\left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \wedge \\ \left(\begin{array}{l} \neg ok \wedge \cdots \wedge (\neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \vee \\ ok' \wedge \cdots \wedge (wait' = wait) \wedge (ref'_{in} = ref_{in}) \end{array} \right) [v_0/v] \wedge wait_0 \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \end{array} \right) \wedge \left(\begin{array}{l} wait \vee \\ \neg wait' \vee \\ \delta \notin ref' \Rightarrow \\ quiescence \end{array} \right) \right)$$

(substitution and prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge \\ (\neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right) \vee \\ \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \wedge \\ (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge (\neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \vee \\ \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \wedge \\ (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right)$$

(substitution)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge (\neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \vee \\ \left(\begin{array}{l} (P \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)))[v_0/v'] \wedge wait_0 \wedge \\ ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right)$$

(def. of $\mathbf{IOCO2}$ and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge (\neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \vee \\ P[v_0/v'] \wedge wait_0 \wedge ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right)$$

(prop. calculus and def. of $\mathbb{I}^{\delta}$ and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}^{\delta}[v_0/v] \vee \\ \mathbf{IOCO2}(P)[v_0/v'] \wedge \neg wait_0 \wedge \mathbf{IOCO2}[v_0/v] \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right)$$

(def. $\mathbf{IOCO2}$ and renaming)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}^{\delta}[v_0/v] \vee \\ \left(\begin{array}{l} P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee (\delta \notin ref_0 \Rightarrow quiescence[v_0/v'])) \wedge \neg wait_0 \wedge \\ Q[v_0/v] \wedge (wait_0 \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \wedge \\ (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}^{\delta}[v_0/v] \vee \left(\begin{array}{l} P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (\neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \wedge \\ (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right) \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}^{\delta}[v_0/v] \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee (\delta \notin ref' \Rightarrow quiescence)) \end{array} \right)$$

(substitution and def. of $\mathbf{IOCO2}$)

$= \exists v_0 \bullet P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}^{\delta}[v_0/v] \vee P[v_0/v'] \wedge \neg wait_0 \wedge \mathbf{IOCO2}(Q)[v_0/v]$ (assumption and def. of $\mathbf{R3}^{\delta}$)

$= \exists v_0 \bullet P[v_0/v'] \wedge \mathbf{R3}^{\delta}(Q)[v_0/v]$ (assumption and def. of ;)

$= P;Q$

Lemma 4.20 (closure-;-IOCO3).

$$\mathbf{IOCO3}(P;Q) = P;Q \text{ provided P and Q are } \mathbf{IOCO3} \text{ and } \mathbf{R3} \text{ healthy}$$

Proof of Lemma 4.20. Similar to the proof of Lemma 4.19.

Lemma 4.21 ($\mathbb{\delta}^{\delta}$-left-zero).

$$\mathbb{\delta}^{\delta};P = \mathbb{\delta}^{\delta}$$

Proof of Lemma 4.21.

$$\begin{array}{lr}
\mathbb{\delta}^{\delta};P = & (\text{closure of } \mathbf{R3}^{\delta}) \\
= \mathbb{I}^{\delta} \lhd wait \rhd (\mathbb{\delta}^{\delta};P) & (\text{def. of } \mathbb{\delta}^{\delta}) \\
= \mathbb{I}^{\delta} \lhd wait \rhd ((\mathbb{I}^{\delta} \lhd wait \rhd tr' - tr \in \delta^* \wedge wait');P) & (\text{def. of } \lhd\rhd) \\
= \mathbb{I}^{\delta} \lhd wait \rhd ((tr' - tr \in \delta^* \wedge wait');P) & (\text{P meets } \mathbf{R3}) \\
= \mathbb{I}^{\delta} \lhd wait \rhd ((tr' - tr \in \delta^* \wedge wait');\mathbb{I}^{\delta} \lhd wait \rhd P) & (\text{P meets } \mathbf{R3}) \\
= \mathbb{I}^{\delta} \lhd wait \rhd ((tr' - tr \in \delta^* \wedge wait');\mathbb{I}^{\delta} \lhd wait \rhd P) & (\text{def. of ;}) \\
= \mathbb{I}^{\delta} \lhd wait \rhd ((tr' - tr \in \delta^* \wedge wait');\mathbb{I}^{\delta}) & (\text{; unit}) \\
= \mathbb{I}^{\delta} \lhd wait \rhd ((tr' - tr \in \delta^* \wedge wait')) & (\text{def. of } \mathbb{\delta}^{\delta}) \\
= \mathbb{\delta}^{\delta} &
\end{array}$$

Lemma 4.22 (unfolded-independent-execution).

$$(P \curlywedge Q);M = \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M[v_0, v_1/0.v, 1.v]$$

Proof of Lemma 4.22.

$$\begin{array}{lr}
(P \curlywedge Q);M) = & (\text{def. of } \curlywedge) \\
= (P;U_0(out\alpha P) \wedge Q;U_1(out\alpha Q));M & (\text{def. of } U_0,\, U_1 \text{ and of ;}) \\
= ((\exists v_0 \bullet P[v_0/v'] \wedge (0.v' = v_0)) \wedge (\exists v_1 \bullet Q[v_1/v'] \wedge (1.v' = v_1))); & M(\text{one point rule}) \\
= (P[0.v'/v'] \wedge Q[1.v'/v']);M & (\text{def. of ;}) \\
= \exists v_0, v_1 \bullet (P[0.v'/v'] \wedge Q[1.v'/v'])[v_0, v_1/0.v', 1.v'] \wedge M[v_0, v_1/0.v, 1.v] & (\text{def. of []}) \\
= \exists v_0, v_1 \bullet (P[v_0/v'] \wedge Q[v_1/v']) \wedge M[v_0, v_1/0.v, 1.v] &
\end{array}$$

Lemma 4.23 (symmetric-M^δ).

$$M^\delta[0.v, 1.v/1.v, 0.v] = M^\delta$$

Proof of Lemma 4.23.

$$\begin{aligned}
& M^\delta[0.v, 1.v/1.v, 0.v] = && \text{(def. of } M^\delta\text{)}\\
&= \left(\begin{array}{r} (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\ ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \\ \neg initQuiet(tr' - tr)) \end{array}\right)[0.v, 1.v/1.v, 0.v] && \text{(def. of [])}\\
&= (1.wait \iff 0.wait) \wedge wait' = 1.wait \wedge (ok' = (1.ok \wedge 0.ok)) \wedge \\
&\quad ((\neg initQuiet(1.tr - tr) \vee \neg initQuiet(0.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) && \text{(prop. calculus)}\\
&= (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
&\quad ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) && \text{(def. of } M^\delta\text{)}\\
&= M^\delta
\end{aligned}$$

Lemma 4.24 (wait-and-ok-M^δ).

$$M^\delta = (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M^\delta$$

Proof of Lemma 4.24.

$$\begin{aligned}
& M^\delta = && \text{(def. of } M^\delta\text{)}\\
&= (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
&\qquad ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) && \text{(prop. calculus)}\\
&= (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
&\qquad (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
&\qquad ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) && \text{(prop. calculus)}\\
&= (0.wait \iff 1.wait) \wedge wait' = (0.wait \vee 1.wait) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
&\qquad (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge (ok' = (0.ok \wedge 1.ok)) \wedge \\
&\qquad ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) && \text{(def. of } M^\delta\text{)}\\
&= (0.wait \iff 1.wait) \wedge wait' = (0.wait \vee 1.wait) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M^\delta
\end{aligned}$$

Lemma 4.25 (symmetric-$M_{\sqcap}^{\eth^{\delta}}$).

$$M_{\sqcap}^{\eth^{\delta}}[0.v,1.v/1.v,0.v] = M_{\sqcap}^{\eth^{\delta}}$$

Proof of Lemma 4.25.

$M_{\sqcap}^{\eth^{\delta}}[0.v,1.v/1.v,0.v] =$ (def. of $M_{\sqcap}^{\eth^{\delta}}$)

$= (M^{\delta} \wedge M_{\sqcap}^{init})[0.v,1.v/1.v,0.v]$ (def. of $M_{\sqcap}^{init}$)

$= \left(M^{\delta} \wedge \left(\begin{array}{l} ((tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))) \vee \\ (tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)))) \end{array}\right)\right)[0.v,1.v/1.v,0.v]$ (def. of [])

$= M^{\delta}[0.v,1.v/1.v,0.v] \wedge \left(\begin{array}{l} ((tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (1.ref \cup 0.ref))) \vee \\ (tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (1.ref \cup 0.ref)))) \end{array}\right)$ (Lemma 4.23)

$= M^{\delta} \wedge \left(\begin{array}{l} ((tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (1.ref \cup 0.ref))) \vee \\ (tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (1.ref \cup 0.ref)))) \end{array}\right)$ (prop. calculus)

$= M^{\delta} \wedge \left(\begin{array}{l} ((tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))) \vee \\ (tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)))) \end{array}\right)$ (def. of $M_{\sqcap}^{init}$)

$= M^{\delta} \wedge M_{\sqcap}^{init}$ (def. of $M_{\sqcap}^{\eth^{\delta}}$)

$= M_{\sqcap}^{\eth^{\delta}}$

Lemma 4.26 (symmetric-$M^{\neg\eth^{\delta}}$).

$$M^{\neg\eth^{\delta}}[0.v,1.v/1.v,0.v] = M^{\neg\eth^{\delta}}$$

Proof of Lemma 4.26.

$M^{\neg\eth^{\delta}}[0.v,1.v/1.v,0.v] =$ (def. of $M^{\neg\eth^{\delta}}$)

$= (M^{\delta} \wedge M^{term})[0.v,1.v/1.v,0.v]$ (def. of M^{term})

$= \left(M^{\delta} \wedge ((tr' = 0.tr \wedge ref' = 0.ref) \vee (tr' = 1.tr \wedge ref' = 1.ref))\right)[0.v,1.v/1.v,0.v]$ (def. of [])

$= M^{\delta}[0.v,1.v/1.v,0.v] \wedge ((tr' = 1.tr \wedge ref' = 1.ref) \vee (tr' = 0.tr \wedge ref' = 0.ref))$ (Lemma 4.23)

$= M^{\delta} \wedge ((tr' = 1.tr \wedge ref' = 1.ref) \vee (tr' = 0.tr \wedge ref' = 0.ref))$ (prop. calculus)

$= M^{\delta} \wedge ((tr' = 0.tr \wedge ref' = 0.ref) \vee (tr' = 1.tr \wedge ref' = 1.ref))$ (def. of M^{term})

$= M^{\delta} \wedge M^{term}$ (def. of $M^{\neg\eth^{\delta}}$)

$= M^{\neg\eth^{\delta}}$

Lemma 4.27 (tr-$M_\sqcap$).

$$M_\sqcap = (tr' = 0.tr \lor tr' = 1.tr) \land M_\sqcap$$

Proof of Lemma 4.27.

$$M_\sqcap = \qquad \text{(def. of } M_\sqcap\text{)}$$

$$= M_\sqcap^{\eth^\delta} \lhd \eth^\delta \rhd M^{\neg \eth^\delta} \qquad \text{(def. of } M_\sqcap^{\eth^\delta}, M^{\neg \eth^\delta}\text{)}$$

$$= (M^\delta \land M_\sqcap^{init}) \lhd \eth^\delta \rhd (M^\delta \land M^{term}) \qquad \text{(def. of } M^{init}, M^{term}\text{)}$$

$$= \left(M^\delta \land \left(\begin{array}{l} tr' = 0.tr \land (ref' = \ldots) \lor \\ tr' = 1.tr \land (ref' = \ldots) \end{array} \right) \right) \lhd \eth^\delta \rhd \left(M^\delta \land \left(\begin{array}{l} tr' = 0.tr \land (ref' = 0.ref) \lor \\ tr' = 1.tr \land (ref' = 1.ref) \end{array} \right) \right)$$

$$\text{(prop. calculus)}$$

$$= \left(M^\delta \land \left(\begin{array}{l} (tr' = 0.tr \lor tr' = 1.tr) \land \\ \left(\begin{array}{l} tr' = 0.tr \land (ref' = \ldots) \lor \\ tr' = 1.tr \land (ref' = \ldots) \end{array} \right) \end{array} \right) \right) \lhd \eth^\delta \rhd \left(M^\delta \land \left(\begin{array}{l} (tr' = 0.tr \lor tr' = 1.tr) \land \\ \left(\begin{array}{l} tr' = 0.tr \land (ref' = 0.ref) \lor \\ tr' = 1.tr \land (ref' = 1.ref) \end{array} \right) \end{array} \right) \right)$$

$$\text{(prop. calculus)}$$

$$= \left(\begin{array}{l} (tr' = 0.tr \lor tr' = 1.tr) \land \\ \left(M^\delta \land \begin{array}{l} tr' = 0.tr \land (ref' = \ldots) \lor \\ tr' = 1.tr \land (ref' = \ldots) \end{array} \right) \end{array} \right) \lhd \eth^\delta \rhd \left(\begin{array}{l} (tr' = 0.tr \lor tr' = 1.tr) \land \\ \left(M^\delta \land \begin{array}{l} tr' = 0.tr \land (ref' = 0.ref) \lor \\ tr' = 1.tr \land (ref' = 1.ref) \end{array} \right) \end{array} \right)$$

$$\text{(def. of } M^{init} \text{ and of } M^{term}\text{)}$$

$$= ((tr' = 0.tr \lor tr' = 1.tr) \land M^\delta \land M_\sqcap^{init}) \lhd \eth^\delta \rhd ((tr' = 0.tr \lor tr' = 1.tr) \land M^\delta \land M^{term}) \qquad \text{(def. of } M_\sqcap^{\eth^\delta}, M^{\neg \eth^\delta}\text{)}$$

$$= ((tr' = 0.tr \lor tr' = 1.tr) \land M_\sqcap^{\eth^\delta}) \lhd \eth^\delta \rhd ((tr' = 0.tr \lor tr' = 1.tr) \land M^{\neg \eth^\delta}) \qquad \text{(distr. of } \land \text{ over if)}$$

$$= (tr' = 0.tr \lor tr' = 1.tr) \land (M_\sqcap^{\eth^\delta} \lhd \eth^\delta \rhd M^{\neg \eth^\delta}) \qquad \text{(def. of } M_\sqcap\text{)}$$

$$= (tr' = 0.tr \lor tr' = 1.tr) \land M_\sqcap$$

Lemma 4.28 (wait-and-ok-$M_\sqcap$).

$$M_\sqcap = (0.wait \iff 1.wait) \land (wait' = (0.wait \lor 1.wait)) \land (ok' = (0.ok \land 1.ok)) \land M_\sqcap$$

Proof of Lemma 4.28.

$$M_\sqcap = \qquad \text{(def. of } M_\sqcap\text{)}$$

$$= M_\sqcap^{\eth^\delta} \lhd \eth^\delta \rhd M^{\neg \eth^\delta} \qquad \text{(def. of } M_\sqcap^{\eth^\delta} \text{ and of } M^{\neg \eth^\delta}\text{)}$$

$$= M^\delta \land M_\sqcap^{init} \lhd \eth^\delta \rhd M^\delta \land M^{term} \qquad \text{(Lemma 4.24)}$$

$$= \left(\begin{array}{l} (0.wait \iff 1.wait) \land \\ (wait' = (0.wait \lor 1.wait)) \land \\ (ok' = (0.ok \land 1.ok)) \land \\ M^\delta \land M_\sqcap^{init} \end{array} \right) \lhd \eth^\delta \rhd \left(\begin{array}{l} (0.wait \iff 1.wait) \land \\ (wait' = (0.wait \lor 1.wait)) \land \\ (ok' = (0.ok \land 1.ok)) \land \\ M^\delta \land M^{term} \end{array} \right) \qquad \text{(distr. of } \land \text{ over if)}$$

$$= \left(\begin{array}{l} (0.wait \iff 1.wait) \land \\ (wait' = (0.wait \lor 1.wait)) \land \\ (ok' = (0.ok \land 1.ok)) \end{array} \right) \land (M^\delta \land M_\sqcap^{init} \lhd \eth^\delta \rhd M^\delta \land M^{term}) \qquad \text{(def. of } M_\sqcap^{\eth^\delta} \text{ and of } M^{\neg \eth^\delta}\text{)}$$

$$= \left(\begin{array}{l} (0.wait \iff 1.wait) \land \\ (wait' = (0.wait \lor 1.wait)) \land \\ (ok' = (0.ok \land 1.ok)) \end{array} \right) \land (M_\sqcap^{\eth^\delta} \lhd \eth^\delta \rhd M^{\neg \eth^\delta}) \qquad \text{(def. of } M_\sqcap\text{)}$$

$$= (0.wait \iff 1.wait) \land (wait' = (0.wait \lor 1.wait)) \land (ok' = (0.ok \land 1.ok)) \land M_\sqcap$$

Lemma 4.29 (wait-and-ref-$M_\sqcap$).

$$
\begin{aligned}
&M_\sqcap = M_\sqcap \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \\
&\left(\left(\begin{array}{l} tr' = 0.tr \wedge ref'_{in} = 0.ref_{in} \wedge ref'_{out} = 0.ref_{out} \wedge \\ ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \vee \\ tr' = 1.tr \wedge ref'_{in} = 1.ref_{in} \wedge ref'_{out} = 1.ref_{out} \wedge \\ ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \end{array}\right) \lhd \mathbb{S}^\delta \rhd \right. \\
&\qquad\qquad \left.\left(\begin{array}{l} tr' = 0.tr \wedge ref' = 0.ref \wedge \\ ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge \\ ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{array}\right)\right)
\end{aligned}
$$

Proof of Lemma 4.29.

$M_\sqcap = M_\sqcap \wedge M_\sqcap =$ (def. of $M_\sqcap$)

$= M_\sqcap \wedge (M_\sqcap^{\mathbb{S}^\delta} \lhd \mathbb{S}^\delta \rhd M^{\neg \mathbb{S}^\delta})$ (def. of $M_\sqcap^{\mathbb{S}^\delta}$ and of $M^{\neg \mathbb{S}^\delta}$)

$= M_\sqcap \wedge ((M^\delta \wedge M_\sqcap^{init}) \lhd \mathbb{S}^\delta \rhd (M^\delta \wedge M^{term}))$ (distr. of $\wedge$ over if)

$= M_\sqcap \wedge (M^\delta \wedge (M_\sqcap^{init} \lhd \mathbb{S}^\delta \rhd M^{term}))$ (Lemma 4.24)

$= M_\sqcap \wedge ((0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M^\delta \wedge (M_\sqcap^{init} \lhd \mathbb{S}^\delta \rhd M^{term}))$

(prop. calculus)

$$
= M_\sqcap \wedge \left(\begin{array}{l} (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \\ (ok' = (0.ok \wedge 1.ok)) \wedge M^\delta \wedge (M_\sqcap^{init} \lhd \mathbb{S}^\delta \rhd M^{term}) \end{array}\right)
$$

(Lemma 4.24)

$= M_\sqcap \wedge ((wait' = (0.wait \wedge 1.wait)) \wedge M^\delta \wedge (M_\sqcap^{init} \lhd \mathbb{S}^\delta \rhd M^{term}))$ (def. of $M_\sqcap^{init}$ and of M^{term})

$$
\begin{aligned}
&= M_\sqcap \wedge (wait' = (0.wait \wedge 1.wait)) \wedge M^\delta \wedge \\
&\left(\left(\begin{array}{l} tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \vee \\ tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \end{array}\right) \lhd \mathbb{S}^\delta \rhd \left(\begin{array}{l} tr' = 0.tr \wedge ref' = 0.ref \vee \\ tr' = 1.tr \wedge ref' = 1.ref \end{array}\right)\right)
\end{aligned}
$$

(def. of ref_{in} and def. of $M_\sqcap$)

$$
\begin{aligned}
&= M_\sqcap \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \\
&\left(\left(\begin{array}{l} tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 0.ref'_{in} = 0.ref' \cap \mathcal{A}_{in} \wedge 1.ref'_{in} = 1.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 0.ref'_{out} = 0.ref' \cap \mathcal{A}_{out} \wedge 1.ref'_{out} = 1.ref' \cap \mathcal{A}_{out} \vee \\ tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 0.ref'_{in} = 0.ref' \cap \mathcal{A}_{in} \wedge 1.ref'_{in} = 1.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 0.ref'_{out} = 0.ref' \cap \mathcal{A}_{out} \wedge 1.ref'_{out} = 1.ref' \cap \mathcal{A}_{out} \end{array}\right) \lhd \mathbb{S}^\delta \rhd \right. \\
&\qquad\qquad \left.\left(\begin{array}{l} tr' = 0.tr \wedge ref' = 0.ref \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 0.ref'_{in} = 0.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 0.ref'_{out} = 0.ref' \cap \mathcal{A}_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 1.ref'_{in} = 1.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 1.ref'_{out} = 1.ref' \cap \mathcal{A}_{out} \end{array}\right)\right)
\end{aligned}
$$

(def. of ref' and $\delta \notin \mathcal{A}_{in}$)

$$
\begin{aligned}
&= M_\sqcap \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \\
&\left(\left(\begin{array}{l} tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{array}\right) \lhd \mathbb{S}^\delta \rhd \right. \\
&\qquad\qquad \left.\left(\begin{array}{l} tr' = 0.tr \wedge ref' = 0.ref \wedge ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{array}\right)\right)
\end{aligned}
$$

Lemma 4.30 (symmetric-$M_\sqcap$).

$$M_\sqcap[0.v, 1.v/1.v, 0.v] = M_\sqcap$$

Proof of Lemma 4.30.

$M_\sqcap[0.v, 1.v/1.v, 0.v] =$ (def. of $M_\sqcap$)

$= (M_\sqcap^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta})[0.v, 1.v/1.v, 0.v]$ (no 1.v and 0.v in $\mathfrak{S}^\delta$)

$= M_\sqcap^{\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v] \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v]$ (Lemma 4.25)

$= M_\sqcap^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v]$ (Lemma 4.26)

$= M_\sqcap^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}$ (def. of $M_\sqcap$)

$= M_\sqcap$

Lemma 4.31 ($M_\sqcap$-reduces-to-skip).

$$M_\sqcap[v_0, v_0/1.v, 0.v] = \mathbb{I}_{\mathrm{rel}}[v_0/v]$$

Proof of Lemma 4.31.

$M_\sqcap[v_0, v_0/0.v, 1.v] =$ (def. of $M_\sqcap$)

$= (M_\sqcap^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta})[v_0, v_0/0.v, 1.v]$ (def. of $M_\sqcap^{\mathfrak{S}^\delta}$ and of $M^{\neg\mathfrak{S}^\delta}$)

$= (M^\delta \wedge M_\sqcap^{init} \lhd \mathfrak{S}^\delta \rhd M^\delta \wedge M^{term})[v_0, v_0/0.v, 1.v]$ (def. of merge relations)

$$= \left(\left(\begin{array}{l} (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge ok' = (0.ok \wedge 1.ok) \wedge \\ ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = 0.tr \wedge ref' = (0.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))) \vee \\ (tr' = 1.tr \wedge ref' = (1.ref \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)))) \end{array} \right) \lhd \mathfrak{S}^\delta \rhd \left(\begin{array}{l} (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge ok' = (0.ok \wedge 1.ok) \wedge \\ ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = 0.tr \wedge ref' = 0.ref) \vee \\ (tr' = 1.tr \wedge ref' = 1.ref)) \end{array} \right) \right) [v_0, v_0/0.v, 1.v]$$

(def. of [])

$$= \left(\begin{array}{l} (wait_0 \iff wait_0) \wedge wait' = wait_0 \wedge ok' = (ok_0 \wedge ok_0) \wedge \\ ((\neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_0 - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = (ref_0 \setminus \{\delta\}) \cup (\{\delta\} \cap (ref_0 \cup ref_0))) \vee \\ (tr' = tr_0 \wedge ref' = (ref_0 \setminus \{\delta\}) \cup (\{\delta\} \cap (ref_0 \cup ref_0)))) \end{array} \right) \lhd \mathfrak{S}^\delta[v_0, v_0/0.v, 1.v] \rhd \left(\begin{array}{l} (wait_0 \iff wait_0) \wedge wait' = wait_0 \wedge ok' = (ok_0 \wedge ok_0) \wedge \\ ((\neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_0 - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = ref_0) \vee \\ (tr' = tr_0 \wedge ref' = ref_0)) \end{array} \right)$$

(prop. calculus)

$$= \left(\begin{array}{l} wait' = wait_0 \wedge ok' = ok_0 \wedge (tr' = tr_0 \wedge ref' = ref_0) \\ (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \end{array} \right) \lhd \mathfrak{S}^\delta[v_0, v_0/0.v, 1.v] \rhd \left(\begin{array}{l} wait' = wait_0 \wedge ok' = ok_0 \wedge (tr' = tr_0 \wedge ref' = ref_0) \\ (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \end{array} \right)$$

(cond. idemp)

$= wait' = wait_0 \wedge ok' = ok_0 \wedge (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge (tr' = tr_0 \wedge ref' = ref_0)$

($tr' = tr_0$ and prop. calculus)

$= wait' = wait_0 \wedge ok' = ok_0 \wedge tr' = tr_0 \wedge ref' = ref_0$ (def. of $\mathbb{I}_{\mathrm{rel}}$)

$= \mathbb{I}_{\mathrm{rel}}[v_0/v]$

Lemma 4.32 (idempotent-$\sqcap^\delta$).

$$P \sqcap^\delta P = P$$

Proof of Lemma 4.32.

$P \sqcap^\delta P =$ (def. of $\sqcap^\delta$)

$= (P \curlywedge P); M_\sqcap$ (Lemma 4.22)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge P[v_1/v'] \wedge M_\sqcap[v_0, v_1/0.v, 1.v]$ (rename variables)

$= \exists v_0 \bullet P[v_0/v'] \wedge P[v_0/v'] \wedge M_\sqcap[v_0, v_0/0.v, 1.v]$ (prop. calculus)

$= \exists v_0 \bullet P[v_0/v'] \wedge M_\sqcap[v_0, v_0/0.v, 1.v]$ (Lemma 4.31)

$= \exists v_0 \bullet P[v_0/v'] \wedge \mathbb{I}_{\text{rel}}[v_0, v]$ (def. of ;)

$= P; \mathbb{I}_{\text{rel}}$ (; unit)

$= P$

Lemma 4.33 (commutative-$\sqcap^\delta$).

$$P \sqcap^\delta Q = Q \sqcap^\delta P$$

Proof of Lemma 4.33.

$P \sqcap^\delta Q =$ (def. of $\sqcap^\delta$)

$= (P \curlywedge Q); M_\sqcap$ (Lemma 4.22)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_\sqcap[v_0, v_1/0.v, 1.v]$ (rename variables)

$= \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge M_\sqcap[v_1, v_0/0.v, 1.v]$ (Lemma 4.30)

$= \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge (M_\sqcap[0.v, 1.v/1.v, 0.v])[v_1, v_0/0.v, 1.v]$ (def. of [])

$= \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge M_\sqcap[v_1, v_0/1.v, 0.v]$ (def. of [])

$= \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge M_\sqcap[v_0, v_1/0.v, 1.v]$ (prop. calculus)

$= \exists v_0, v_1 \bullet Q[v_0/v'] \wedge P[v_1/v'] \wedge M_\sqcap[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (Q \curlywedge P); M_\sqcap$ (def. of $\sqcap^\delta$)

$= Q \sqcap^\delta P$

Lemma 4.34 (closure-$\sqcap^{\delta}$-R1).

$$\mathbf{R1}(P \ \sqcap^{\delta} \ Q) = P \ \sqcap^{\delta} \ Q \text{ provided P and Q are } \mathbf{R1} \text{ healthy}$$

Proof of Lemma 4.34.

$\mathbf{R1}(P \ \sqcap^{\delta} \ Q) =$ (assumption)

$= \mathbf{R1}(\mathbf{R1}(P) \ \sqcap^{\delta} \ \mathbf{R1}(Q)) =$ (def. of **R1**)

$= ((P \wedge (tr \le tr')) \ \sqcap^{\delta} \ (Q \wedge (tr \le tr'))) \wedge (tr \le tr')$ (def. of $\curlywedge$)

$= (((P \wedge (tr \le tr')) \curlywedge (Q \wedge (tr \le tr'))); M_{\sqcap}) \wedge (tr \le tr')$ (Lemma 4.22)

$= (\exists v_0, v_1 \bullet (P \wedge (tr \le tr'))[v_0/v'] \wedge (Q \wedge (tr \le tr'))[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) \wedge (tr \le tr')$ (def. of [])

$= (\exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \le tr_0) \wedge Q[v_1/v'] \wedge (tr \le tr_1) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) \wedge (tr \le tr')$ (Lemma 4.27)

$= (\exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \le tr_0) \wedge Q[v_1/v'] \wedge (tr \le tr_1) \wedge$
$\quad (M_{\sqcap} \wedge (tr' = 0.tr \vee tr' = 1.tr))[v_0, v_1/0.v, 1.v]) \wedge (tr \le tr')$ (def. of [])

$= (\exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \le tr_0) \wedge Q[v_1/v'] \wedge (tr \le tr_1) \wedge$
$\quad M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr' = tr_0 \vee tr' = tr_1)) \wedge (tr \le tr')$ (prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \le tr_0) \wedge Q[v_1/v'] \wedge (tr \le tr_1) \wedge$
$\quad M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr' = tr_0 \vee tr' = tr_1) \wedge (tr \le tr')$ (prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr \le tr_0) \wedge (tr \le tr_1) \wedge (tr' = tr_0) \wedge (tr \le tr') \vee$
$\quad P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr \le tr_0) \wedge (tr \le tr_1) \wedge (tr' = tr_1) \wedge (tr \le tr')$
(properties of =)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr \le tr_0) \wedge (tr \le tr_1) \wedge (tr' = tr_0) \vee$
$\quad P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr \le tr_0) \wedge (tr \le tr_1) \wedge (tr' = tr_1)$ (prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (tr \le tr_0) \wedge (tr \le tr_1) \wedge ((tr' = tr_0) \vee (tr' = tr_1))$
(prop. calculus and def. of [])

$= \exists v_0, v_1 \bullet (P \wedge (tr \le tr'))[v_0/v'] \wedge (Q \wedge (tr \le tr'))[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (def. of **R1**)

$= \exists v_0, v_1 \bullet (\mathbf{R1}(P))[v_0/v'] \wedge (\mathbf{R1}(Q))[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q); M_{\sqcap}$ (def. of $\sqcap^{\delta}$)

$= P \ \sqcap^{\delta} \ Q$

Lemma 4.35 (closure-$\sqcap^{\delta}$-R2).

$$\mathbf{R2}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided P and Q are } \mathbf{R2} \text{ healthy}$$

Proof of Lemma 4.35.

$\mathbf{R2}(P(tr,tr') \sqcap^{\delta} Q(tr,tr')) =$ (def. of **R2**)

$= (P(tr,tr') \sqcap^{\delta} Q(tr,tr'))(\langle\rangle, tr' - tr))$ (def. of $\sqcap^{\delta}$)

$= ((P(tr,tr') \curlywedge Q(tr,tr')); M_{\sqcap}(tr,tr'))(\langle\rangle, tr' - tr))$ (Lemma 4.22)

$= (\exists v_0, v_1 \bullet P(tr,tr')[v_0/v'] \wedge Q(tr,tr')[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v](tr,tr'))(\langle\rangle, tr' - tr))$ (assumption)

$= (\exists v_0, v_1 \bullet \mathbf{R2}(P(tr,tr'))[v_0/v'] \wedge \mathbf{R2}(Q)(tr,tr')[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v](tr,tr'))(\langle\rangle, tr' - tr))$ (def. of **R2**)

$= (\exists v_0, v_1 \bullet P(\langle\rangle, tr' - tr)[v_0/v'] \wedge Q(\langle\rangle, tr' - tr)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v](tr,tr'))(\langle\rangle, tr' - tr)$ (substitution)

$= (\exists v_0, v_1 \bullet P[v_0/v'](\langle\rangle, tr_0 - tr) \wedge Q[v_1/v'](\langle\rangle, tr_1 - tr) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v](tr,tr'))(\langle\rangle, tr' - tr)$

(def. of $M_{\sqcap}$ and substitution)

$$= \exists v_0, v_1 \bullet P[v_0/v'](\langle\rangle, tr_0 - tr) \wedge Q[v_1/v'](\langle\rangle, tr_1 - tr) \wedge$$
$$\left(\left(\begin{array}{l} ((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\ \neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_1 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = \ldots) \vee (tr' = tr_1 \wedge ref' = dots)) \end{array}\right) \lhd \mathfrak{S}^{\delta} \rhd \right.$$
$$\left.\left(\begin{array}{l} ((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\ \neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_1 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = \ldots) \vee (tr' = tr_1 \wedge ref' = dots)) \end{array}\right)\right)(\langle\rangle, tr' - tr)$$

(substitution)

$$= \exists v_0, v_1 \bullet P[v_0/v'](\langle\rangle, tr_0 - tr) \wedge Q[v_1/v'](\langle\rangle, tr_1 - tr) \wedge$$
$$\left(\left(\begin{array}{l} ((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\ \neg initQuiet((tr_0 - tr) - \langle\rangle) \vee \neg initQuiet((tr_1 - tr) - \langle\rangle) \Rightarrow \neg initQuiet((tr' - tr) - \langle\rangle)) \wedge \\ ((tr' - tr = tr_0 - tr \wedge ref' = \ldots) \vee (tr' - tr = tr_1 - tr \wedge ref' = dots)) \end{array}\right) \lhd \mathfrak{S}^{\delta} \rhd \right.$$
$$\left.\left(\begin{array}{l} ((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\ \neg initQuiet((tr_0 - tr) - \langle\rangle) \vee \neg initQuiet((tr_1 - tr) - \langle\rangle) \Rightarrow \neg initQuiet((tr' - tr) - \langle\rangle)) \wedge \\ ((tr' - tr = tr_0 - tr \wedge ref' = \ldots) \vee (tr' - tr = tr_1 - tr \wedge ref' = dots)) \end{array}\right)\right)$$

(prop. calculus)

$$= \exists v_0, v_1 \bullet P[v_0/v'](\langle\rangle, tr_0 - tr) \wedge Q[v_1/v'](\langle\rangle, tr_1 - tr) \wedge$$
$$\left(\left(\begin{array}{l} ((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\ \neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_1 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = \ldots) \vee (tr' = tr_1 \wedge ref' = dots)) \end{array}\right) \lhd \mathfrak{S}^{\delta} \rhd \right.$$
$$\left.\left(\begin{array}{l} ((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\ \neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_1 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = \ldots) \vee (tr' = tr_1 \wedge ref' = dots)) \end{array}\right)\right)$$

(def. of $M_{\sqcap}$ and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'](\langle\rangle, tr_0 - tr) \wedge Q[v_1/v'](\langle\rangle, tr_1 - tr) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (substitution)

$= \exists v_0, v_1 \bullet (P(\langle\rangle, tr' - tr))[v_0/v'] \wedge (Q(\langle\rangle, tr' - tr))[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (def. of **R2**)

$= \exists v_0, v_1 \bullet \mathbf{R2}(P(tr,tr'))[v_0/v'] \wedge \mathbf{R2}(Q(tr,tr'))[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P(tr,tr')[v_0/v'] \wedge Q(tr,tr')[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P(tr,tr') \curlywedge Q(tr,tr')); M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P(tr,tr') \curlywedge Q(tr,tr')); M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (def. of $\sqcap^{\delta}$)

$= P(tr,tr') \sqcap^{\delta} Q(tr,tr')$

Lemma 4.36 (closure-$\sqcap^{\delta}$-$\mathbf{R3}^{\delta}$).

$$\mathbf{R3}^{\delta}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided P and Q are } \mathbf{R3}^{\delta} \text{ healthy}$$

Proof of Lemma 4.36.

$$\begin{aligned}
&\mathbf{R3}^{\delta}(P \sqcap^{\delta} Q) = && \text{(def. of } \mathbf{R3}^{\delta}\text{)}\\
&= \mathbb{I}^{\delta} \lhd wait \rhd (P \sqcap^{\delta} Q) && \text{(Lemma 4.32)}\\
&= (\mathbb{I}^{\delta} \sqcap^{\delta} \mathbb{I}^{\delta}) \lhd wait \rhd (P \sqcap^{\delta} Q) && \text{(def. of } \sqcap^{\delta}\text{)}\\
&= (\mathbb{I}^{\delta} \curlywedge \mathbb{I}^{\delta}); M_{\sqcap} \lhd wait \rhd (P \curlywedge Q); M_{\sqcap} && \text{(Lemma 4.22)}\\
&= (\exists v_0, v_1 \bullet \mathbb{I}^{\delta}[v_0/v'] \wedge \mathbb{I}^{\delta}[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) \lhd wait \rhd\\
&\qquad (\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) && \text{(wait not quantified)}\\
&= \exists v_0, v_1 \bullet ((\mathbb{I}^{\delta}[v_0/v'] \wedge \mathbb{I}^{\delta}[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) \lhd wait \rhd\\
&\qquad (P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v])) && \text{(}\wedge\text{-if distr.)}\\
&= \exists v_0, v_1 \bullet ((\mathbb{I}^{\delta}[v_0/v'] \wedge \mathbb{I}^{\delta}[v_1/v']) \lhd wait \rhd\\
&\qquad (P[v_0/v'] \wedge Q[v_1/v'])) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] && \text{(mutual distribution)}\\
&= \exists v_0, v_1 \bullet (\mathbb{I}^{\delta}[v_0/v'] \lhd wait \rhd P[v_0/v']) \wedge (\mathbb{I}^{\delta}[v_1/v'] \lhd wait \rhd Q[v_1/v']) \wedge\\
&\qquad M_{\sqcap}[v_0, v_1/0.v, 1.v] && \text{(distr. of [])}\\
&= \exists v_0, v_1 \bullet (\mathbb{I}^{\delta} \lhd wait \rhd P)[v_0/v'] \wedge (\mathbb{I}^{\delta} \lhd wait \rhd Q)[v_1/v'] \wedge\\
&\qquad M_{\sqcap}[v_0, v_1/0.v, 1.v] && \text{(def. of } \mathbf{R3}^{\delta}\text{)}\\
&= \exists v_0, v_1 \bullet \mathbf{R3}^{\delta} P[v_0/v'] \wedge \mathbf{R3}^{\delta}(Q)[v_1/v'] M_{\sqcap}[v_0, v_1/0.v, 1.v] && \text{(assumption)}\\
&= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] M_{\sqcap}[v_0, v_1/0.v, 1.v] && \text{(Lemma 4.22)}\\
&= (P \curlywedge Q); M_{\sqcap} && \text{(def. of } \sqcap^{\delta}\text{)}\\
&= P \sqcap^{\delta} Q
\end{aligned}$$

Lemma 4.37 (closure-$\sqcap^{\delta}$-IOCO1).

$$\mathbf{IOCO1}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided P and Q are } \mathbf{IOCO1} \text{ healthy}$$

Proof of Lemma 4.37.

$$
\begin{array}{ll}
\mathbf{IOCO1}(P \sqcap^{\delta} Q) = & \text{(def. } \sqcap^{\delta}\text{)} \\
= \mathbf{IOCO1}(P \curlywedge Q; M_{\sqcap}) & \text{(Lemma 4.22)} \\
= \mathbf{IOCO1}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) & \text{(def. } \mathbf{IOCO1}\text{)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (ok \Rightarrow (wait' \vee ok')) & \text{(Lemma 4.28)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge \left(\begin{array}{l} (0.wait \iff 1.wait) \wedge \\ (wait' = (0.wait \vee 1.wait)) \wedge \\ (ok' = (0.ok \wedge 1.ok)) \wedge M_{\sqcap} \end{array} \right) [v_0, v_1/0.v, 1.v] \wedge (ok \Rightarrow (wait' \vee ok')) & \\
& \text{(def. of [])} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge (ok \Rightarrow (wait' \vee ok')) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge & \\
\qquad (ok \Rightarrow (wait_0 \vee wait_1 \vee (ok_0 \wedge ok_1))) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0 \vee wait_1)) \wedge (ok \Rightarrow (ok_1 \vee wait_0 \vee wait_1)) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0)) \wedge (ok \Rightarrow (ok_1 \vee wait_1)) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge & \\
\qquad ((0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M_{\sqcap})[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0)) \wedge (ok \Rightarrow (ok_1 \vee wait_1)) & \text{(Lemma 4.28)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0)) \wedge (ok \Rightarrow (ok_1 \vee wait_1)) & \text{(def. of [])} \\
= \exists v_0, v_1 \bullet (P \wedge (ok \Rightarrow (ok' \vee wait')))[v_0/v'] \wedge & \\
\qquad (Q \wedge (ok \Rightarrow (ok' \vee wait')))[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] & \text{(def. } \mathbf{IOCO1}\text{)} \\
= \exists v_0, v_1 \bullet \mathbf{IOCO1}(P)[v_0/v'] \wedge \mathbf{IOCO1}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] & \text{(assumption)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] & \text{(Lemma 4.22)} \\
= P \curlywedge Q; M_{\sqcap} & \text{(def. of } \sqcap^{\delta}\text{)} \\
= P \sqcap^{\delta} Q &
\end{array}
$$

Lemma 4.38 (closure-$\sqcap^{\delta}$-IOCO2).

$$\mathbf{IOCO2}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided P and Q are } \mathbf{IOCO2} \text{ healthy}$$

Proof of Lemma 4.38.

$\mathbf{IOCO2}(P \sqcap^{\delta} Q) =$ (def. of $\sqcap^{\delta}$)

$= \mathbf{IOCO2}((P \curlywedge Q); M_{\sqcap})$ (Lemma 4.22)

$= \mathbf{IOCO2}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v])$ (def. of **IOCO2**)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$

(assumption)

$= \exists v_0, v_1 \bullet \mathbf{IOCO2}(P)[v_0/v'] \wedge \mathbf{IOCO2}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge$
$(\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. **IOCO2** and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge Q[v_1/v'] \wedge$
$(\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge$
$(\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow (\neg wait' \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref'))))$

(Lemma 4.29 and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge Q[v_1/v'] \wedge$
$(\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge$
$(\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow (\neg wait' \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref')))) \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$
$$\left(\begin{pmatrix} tr' = tr_0 \wedge \dots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \dots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \eth^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \dots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \dots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$$

(prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge Q[v_1/v'] \wedge$
$(\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$
$$\left(\begin{pmatrix} tr' = tr_0 \wedge \dots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \dots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \eth^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \dots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \dots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$$

(substitution and Lemma 4.29)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge$
$Q[v_1/v'] \wedge (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$

(substitution and def. of **IOCO2**)

$= \exists v_0, v_1 \bullet \mathbf{IOCO2}(P)[v_0/v'] \wedge \mathbf{IOCO2}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q); M_{\sqcap}$ (def. of $\sqcap^{\delta}$)

$= P \sqcap^{\delta} Q$

Lemma 4.39 (closure-$\sqcap^{\delta}$-IOCO3).

$$\textbf{IOCO3}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided P and Q are } \textbf{IOCO3} \text{ healthy}$$

Proof of Lemma 4.39.

$$\textbf{IOCO3}(P \sqcap^{\delta} Q) = \qquad \text{(def. of } \sqcap^{\delta}\text{)}$$

$$= \textbf{IOCO3}((P \curlywedge Q) ; M_{\sqcap}) \qquad \text{(Lemma 4.22)}$$

$$= \textbf{IOCO3}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) \qquad \text{(def. of } \textbf{IOCO3}\text{)}$$

$$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s^\frown \delta^*)))$$

(assumption)

$$\begin{aligned}= \exists v_0, v_1 \bullet & \textbf{IOCO3}(P)[v_0/v'] \wedge \textbf{IOCO3}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge \\ & (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s^\frown \delta^*)))\end{aligned}$$

(def. **IOCO3** and substitution)

$$\begin{aligned}= \exists v_0, v_1 \bullet & P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s^\frown \delta^*))) \wedge Q[v_1/v'] \wedge \\ & (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s^\frown \delta^*))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge \\ & (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s^\frown \delta^*)))\end{aligned}$$

(Lemma 4.29 and substitution)

$$\begin{aligned}= \exists v_0, v_1 \bullet & P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s^\frown \delta^*))) \wedge Q[v_1/v'] \wedge \\ & (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s^\frown \delta^*))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge \\ & (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s^\frown \delta^*))) \wedge \\ & (wait' = (wait_0 \wedge wait_1)) \wedge \\ & \left(\left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{array} \right) \lhd \delta^{\delta} \rhd \left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{array} \right) \right)\end{aligned}$$

(prop. calculus)

$$\begin{aligned}= \exists v_0, v_1 \bullet & P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s^\frown \delta^*))) \wedge Q[v_1/v'] \wedge \\ & (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s^\frown \delta^*))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge \\ & (wait' = (wait_0 \wedge wait_1)) \wedge \\ & (wait' = (wait_0 \wedge wait_1)) \wedge \\ & \left(\left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{array} \right) \lhd \delta^{\delta} \rhd \left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{array} \right) \right)\end{aligned}$$

(substitution and Lemma 4.29)

$$\begin{aligned}= \exists v_0, v_1 \bullet & P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s^\frown \delta^*))) \wedge \\ & \quad Q[v_1/v'] \wedge (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s^\frown \delta^*))) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]\end{aligned}$$

(substitution and def. of **IOCO3**)

$$= \exists v_0, v_1 \bullet \textbf{IOCO3}(P)[v_0/v'] \wedge \textbf{IOCO3}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \qquad \text{(assumption)}$$

$$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \qquad \text{(Lemma 4.22)}$$

$$= (P \curlywedge Q) ; M_{\sqcap} \qquad \text{(def. of } \sqcap^{\delta}\text{)}$$

$$= P \sqcap^{\delta} Q$$

Lemma 4.40 (tr-M_+).

$$M_+ = (tr' = 0.tr \vee tr' = 1.tr) \wedge M_+$$

Proof of Lemma 4.40.

$$
\begin{aligned}
& M_+ = && \text{(def. of } M_+\text{)}\\
&= M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta} && \text{(def. of } M_+^{\mathfrak{S}^\delta}, M^{\neg\mathfrak{S}^\delta}\text{)}\\
&= (M^\delta \wedge M_+^{init}) \lhd \mathfrak{S}^\delta \rhd (M^\delta \wedge M^{term}) && \text{(def. of } M^{init}, M^{term}\text{)}\\
&= \left(M^\delta \wedge \begin{pmatrix} (ref' = \ldots)\wedge \\ (tr' = 0.tr \vee tr' = 1.tr) \end{pmatrix}\right) \lhd \mathfrak{S}^\delta \rhd \left(M^\delta \wedge \begin{pmatrix} tr' = 0.tr \wedge (ref' = 0.ref)\vee \\ tr' = 1.tr \wedge (ref' = 1.ref) \end{pmatrix}\right) && \\
& && \text{(prop. calculus)}\\
&= \left(M^\delta \wedge \begin{pmatrix} (tr' = 0.tr \vee tr' = 1.tr)\wedge \\ \begin{pmatrix} (ref' = \ldots)\wedge \\ (tr' = 0.tr \vee tr' = 1.tr) \end{pmatrix} \end{pmatrix}\right) \lhd \mathfrak{S}^\delta \rhd \left(M^\delta \wedge \begin{pmatrix} (tr' = 0.tr \vee tr' = 1.tr)\wedge \\ \begin{pmatrix} tr' = 0.tr \wedge (ref' = 0.ref)\vee \\ tr' = 1.tr \wedge (ref' = 1.ref) \end{pmatrix} \end{pmatrix}\right) && \\
& && \text{(prop. calculus)}\\
&= \begin{pmatrix} (tr' = 0.tr \vee tr' = 1.tr)\wedge \\ \left(M^\delta \wedge \begin{matrix} (ref' = \ldots)\wedge \\ (tr' = 0.tr \vee tr' = 1.tr) \end{matrix}\right) \end{pmatrix} \lhd \mathfrak{S}^\delta \rhd \begin{pmatrix} (tr' = 0.tr \vee tr' = 1.tr)\wedge \\ \left(M^\delta \wedge \begin{matrix} tr' = 0.tr \wedge (ref' = 0.ref)\vee \\ tr' = 1.tr \wedge (ref' = 1.ref) \end{matrix}\right) \end{pmatrix} && \\
& && \text{(def. of } M^{init} \text{ and of } M^{term}\text{)}\\
&= ((tr' = 0.tr \vee tr' = 1.tr) \wedge M^\delta \wedge M_+^{init}) \lhd \mathfrak{S}^\delta \rhd ((tr' = 0.tr \vee tr' = 1.tr) \wedge M^\delta \wedge M^{term}) && \text{(def. of } M_+^{\mathfrak{S}^\delta}, M^{\neg\mathfrak{S}^\delta}\text{)}\\
&= ((tr' = 0.tr \vee tr' = 1.tr) \wedge M_+^{\mathfrak{S}^\delta}) \lhd \mathfrak{S}^\delta \rhd ((tr' = 0.tr \vee tr' = 1.tr) \wedge M^{\neg\mathfrak{S}^\delta}) && \text{(distr. of } \wedge \text{ over if)}\\
&= (tr' = 0.tr \vee tr' = 1.tr) \wedge (M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}) && \text{(def. of } M_+\text{)}\\
&= (tr' = 0.tr \vee tr' = 1.tr) \wedge M_+
\end{aligned}
$$

Lemma 4.41 (wait-and-ok-M_+).

$$M_+ = (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M_+$$

Proof of Lemma 4.41.

$$
\begin{aligned}
& M_+ = && \text{(def. of } M_+\text{)}\\
&= M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta} && \text{(def. of } M_+^{\mathfrak{S}^\delta} \text{ and of } M^{\neg\mathfrak{S}^\delta}\text{)}\\
&= M^\delta \wedge M_+^{init} \lhd \mathfrak{S}^\delta \rhd M^\delta \wedge M^{term} && \text{(Lemma 4.24)}\\
&= \begin{pmatrix} (0.wait \iff 1.wait)\wedge \\ (wait' = (0.wait \vee 1.wait))\wedge \\ (ok' = (0.ok \wedge 1.ok))\wedge \\ M^\delta \wedge M_+^{init} \end{pmatrix} \lhd \mathfrak{S}^\delta \rhd \begin{pmatrix} (0.wait \iff 1.wait)\wedge \\ (wait' = (0.wait \vee 1.wait))\wedge \\ (ok' = (0.ok \wedge 1.ok))\wedge \\ M^\delta \wedge M^{term} \end{pmatrix} && \text{(distr. of } \wedge \text{ over if)}\\
&= \begin{pmatrix} (0.wait \iff 1.wait)\wedge \\ (wait' = (0.wait \vee 1.wait))\wedge \\ (ok' = (0.ok \wedge 1.ok)) \end{pmatrix} \wedge (M^\delta \wedge M_+^{init} \lhd \mathfrak{S}^\delta \rhd M^\delta \wedge M^{term}) && \text{(def. of } M_+^{\mathfrak{S}^\delta} \text{ and of } M^{\neg\mathfrak{S}^\delta}\text{)}\\
&= \begin{pmatrix} (0.wait \iff 1.wait)\wedge \\ (wait' = (0.wait \vee 1.wait))\wedge \\ (ok' = (0.ok \wedge 1.ok)) \end{pmatrix} \wedge (M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}) && \text{(def. of } M_+\text{)}\\
&= (0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M_+
\end{aligned}
$$

Lemma 4.42 (wait-and-ref-M_+).

$$M_+ = M_+ \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \left(\begin{pmatrix} tr' = 0.tr \wedge ref'_{in} = 0.ref_{in} \wedge ref'_{out} = 0.ref_{out} \wedge \\ ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \vee \\ tr' = 1.tr \wedge ref'_{in} = 1.ref_{in} \wedge ref'_{out} = 1.ref_{out} \wedge \\ ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \end{pmatrix} \lhd \mathbb{S}^{\delta} \rhd \begin{pmatrix} tr' = 0.tr \wedge ref' = 0.ref \wedge \\ ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge \\ ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{pmatrix} \right)$$

Proof of Lemma 4.42.

$M_+ = M_+ \wedge M_+ =$ (def. of M_+)

$= M_+ \wedge (M_+^{\mathbb{S}^\delta} \lhd \mathbb{S}^\delta \rhd M^{\neg \mathbb{S}^\delta})$ (def. of $M_+^{\mathbb{S}^\delta}$ and of $M^{\neg\mathbb{S}^\delta}$)

$= M_+ \wedge ((M^\delta \wedge M_+^{init}) \lhd \mathbb{S}^\delta \rhd (M^\delta \wedge M^{term}))$ (distr. of $\wedge$ over if)

$= M_+ \wedge (M^\delta \wedge (M_+^{init} \lhd \mathbb{S}^\delta \rhd M^{term}))$ (Lemma 4.24)

$= M_+ \wedge ((0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M^\delta \wedge (M_+^{init} \lhd \mathbb{S}^\delta \rhd M^{term}))$

(prop. calculus)

$= M_+ \wedge ((0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (wait' = (0.wait \wedge 1.wait)) \wedge$
$(ok' = (0.ok \wedge 1.ok)) \wedge M^\delta \wedge (M_+^{init} \lhd \mathbb{S}^\delta \rhd M^{term}))$

(Lemma 4.24)

$= M_+ \wedge (wait' = (0.wait \wedge 1.wait)) \wedge M^\delta \wedge (M_+^{init} \lhd \mathbb{S}^\delta \rhd M^{term})$ (def. of M_+^{init} and of M^{term})

$$= M_+ \wedge (wait' = (0.wait \wedge 1.wait)) \wedge M^\delta \wedge \left(\begin{pmatrix} tr' = 0.tr \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \vee \\ tr' = 1.tr \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \end{pmatrix} \lhd \mathbb{S}^\delta \rhd \begin{pmatrix} tr' = 0.tr \wedge ref' = 0.ref \vee \\ tr' = 1.tr \wedge ref' = 1.ref \end{pmatrix} \right)$$

(def. of ref_{in} and def. of M_+)

$$= M_+ \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \left(\begin{pmatrix} tr' = 0.tr \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 0.ref'_{in} = 0.ref' \cap \mathcal{A}_{in} \wedge 1.ref'_{in} = 1.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 0.ref'_{out} = 0.ref' \cap \mathcal{A}_{out} \wedge 1.ref'_{out} = 1.ref' \cap \mathcal{A}_{out} \vee \\ tr' = 1.tr \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 0.ref'_{in} = 0.ref' \cap \mathcal{A}_{in} \wedge 1.ref'_{in} = 1.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 0.ref'_{out} = 0.ref' \cap \mathcal{A}_{out} \wedge 1.ref'_{out} = 1.ref' \cap \mathcal{A}_{out} \end{pmatrix} \lhd \mathbb{S}^\delta \rhd \begin{pmatrix} tr' = 0.tr \wedge ref' = 0.ref \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 0.ref'_{in} = 0.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 0.ref'_{out} = 0.ref' \cap \mathcal{A}_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge \\ \quad ref'_{in} = ref' \cap \mathcal{A}_{in} \wedge 1.ref'_{in} = 1.ref' \cap \mathcal{A}_{in} \wedge \\ \quad ref'_{out} = ref' \cap \mathcal{A}_{out} \wedge 1.ref'_{out} = 1.ref' \cap \mathcal{A}_{out} \end{pmatrix} \right)$$

(def. of ref' and $\delta \notin \mathcal{A}_{in}$)

$$= M^\delta \wedge (wait' = (0.wait \wedge 1.wait)) \wedge \left(\begin{pmatrix} tr' = 0.tr \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge \\ \quad ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)) \wedge \\ \quad ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{pmatrix} \lhd \mathbb{S}^\delta \rhd \begin{pmatrix} tr' = 0.tr \wedge ref' = 0.ref \wedge ref'_{in} = 0.ref'_{in} \wedge ref'_{out} = 0.ref'_{out} \vee \\ tr' = 1.tr \wedge ref' = 1.ref \wedge ref'_{in} = 1.ref'_{in} \wedge ref'_{out} = 1.ref'_{out} \end{pmatrix} \right)$$

Lemma 4.43 (symmetric-M_+).

$$M_+[0.v, 1.v/1.v, 0.v] = M_+$$

Proof of Lemma 4.43.

$$\begin{aligned}
& M_+[0.v, 1.v/1.v, 0.v] = && \text{(def. of } M_+\text{)}\\
& = (M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta})[0.v, 1.v/1.v, 0.v] && \text{(no 1.v and 0.v in } \mathfrak{S}^\delta\text{)}\\
& = M_+^{\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v] \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v] && \text{(Lemma 4.25)}\\
& = M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v] && \text{(Lemma 4.26)}\\
& = M_+^{\mathfrak{S}^\delta} \lhd \mathfrak{S}^\delta \rhd M^{\neg\mathfrak{S}^\delta} && \text{(def. of } M_+\text{)}\\
& = M_+
\end{aligned}$$

Lemma 4.44 (symmetric-$M_+^{\mathfrak{S}^\delta}$).

$$M_+^{\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v] = M_+^{\mathfrak{S}^\delta}$$

Proof of Lemma 4.44.

$$\begin{aligned}
& M_+^{\mathfrak{S}^\delta}[0.v, 1.v/1.v, 0.v] = && \text{(def. of } M_+^{\mathfrak{S}^\delta}\text{)}\\
& = (M^\delta \wedge M_+^{init})[0.v, 1.v/1.v, 0.v] && \text{(def. of } M_+^{init}\text{)}\\
& = \left(M^\delta \wedge \left(\begin{array}{l} (ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)))\wedge \\ (tr' = 0.tr \vee tr' = 1.tr) \end{array}\right)\right)[0.v, 1.v/1.v, 0.v] && \text{(def. of [])}\\
& = M^\delta[0.v, 1.v/1.v, 0.v] \wedge \left(\begin{array}{l} (ref' = ((1.ref \cap 0.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (1.ref \cup 0.ref)))\wedge \\ (tr' = 1.tr \vee tr' = 0.tr) \end{array}\right) && \text{(Lemma 4.23)}\\
& = M^\delta \wedge \left(\begin{array}{l} (ref' = ((1.ref \cap 0.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (1.ref \cup 0.ref)))\wedge \\ (tr' = 1.tr \vee tr' = 0.tr) \end{array}\right) && \text{(symmetry of } \cap, \vee\text{)}\\
& = M^\delta \wedge \left(\begin{array}{l} (ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref)))\wedge \\ (tr' = 0.tr \vee tr' = 1.tr) \end{array}\right) && \text{(def. of } M_+^{init}\text{)}\\
& = M^\delta \wedge M_+^{init} && \text{(def. of } M_+^{\mathfrak{S}^\delta}\text{)}\\
& = M_+^{\mathfrak{S}^\delta}
\end{aligned}$$

Lemma 4.45 (M_+-reduces-to-skip).

$$M_+[v_0, v_0/1.v, 0.v] = \mathbb{I}_{\text{rel}}[v_0/v]$$

Proof of Lemma 4.45.

$M_+[v_0, v_0/0.v, 1.v] =$ (def. of M_+)

$= (M_+^{\mathbb{S}^\delta} \lhd \mathbb{S}^\delta \rhd M^{\neg\mathbb{S}^\delta})[v_0, v_0/0.v, 1.v]$ (def. of $M_+^{\mathbb{S}^\delta}$ and of $M^{\neg\mathbb{S}^\delta}$)

$= (M^\delta \wedge M_+^{init} \lhd \mathbb{S}^\delta \rhd M^\delta \wedge M^{term})[v_0, v_0/0.v, 1.v]$ (def. of merge relations)

$$= \left(\left(\begin{array}{l} (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge ok' = (0.ok \wedge 1.ok) \wedge \\ ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = 0.tr \vee tr' = 1.tr) \wedge ref' = ((0.ref \cap 1.ref) \setminus \{\delta\}) \cup (\{\delta\} \cap (0.ref \cup 1.ref))) \end{array}\right) \lhd \mathbb{S}^\delta \rhd \left(\begin{array}{l} (0.wait \iff 1.wait) \wedge wait' = 0.wait \wedge ok' = (0.ok \wedge 1.ok) \wedge \\ ((\neg initQuiet(0.tr - tr) \vee \neg initQuiet(1.tr - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = 0.tr \wedge ref' = 0.ref) \vee \\ (tr' = 1.tr \wedge ref' = 1.ref)) \end{array}\right)\right)[v_0, v_0/0.v, 1.v]$$

(def. of [])

$$= \left(\begin{array}{l} (wait_0 \iff wait_0) \wedge wait' = wait_0 \wedge ok' = (ok_0 \wedge ok_0) \wedge \\ ((\neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_0 - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \vee tr' = tr_0) \wedge ref' = ((ref_0 \cap ref_0) \setminus \{\delta\}) \cup (\{\delta\} \cap (ref_0 \cup ref_0))) \end{array}\right) \lhd \mathbb{S}^\delta[v_0, v_0/0.v, 1.v] \rhd \left(\begin{array}{l} (wait_0 \iff wait_0) \wedge wait' = wait_0 \wedge ok' = (ok_0 \wedge ok_0) \wedge \\ ((\neg initQuiet(tr_0 - tr) \vee \neg initQuiet(tr_0 - tr)) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ ((tr' = tr_0 \wedge ref' = ref_0) \vee \\ (tr' = tr_0 \wedge ref' = ref_0)) \end{array}\right)$$

(prop. calculus)

$$= \left(\begin{array}{l} wait' = wait_0 \wedge ok' = ok_0 \wedge \\ (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ (tr' = tr_0 \wedge ref' = ref_0) \end{array}\right) \lhd \mathbb{S}^\delta[v_0, v_0/0.v, 1.v] \rhd \left(\begin{array}{l} wait' = wait_0 \wedge ok' = ok_0 \wedge \\ (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge \\ (tr' = tr_0 \wedge ref' = ref_0) \end{array}\right)$$

(cond. idemp)

$= wait' = wait_0 \wedge ok' = ok_0 \wedge (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr' - tr)) \wedge (tr' = tr_0 \wedge ref' = ref_0)$

$(tr' = tr_0)$

$= wait' = wait_0 \wedge ok' = ok_0 \wedge (\neg initQuiet(tr_0 - tr) \Rightarrow \neg initQuiet(tr_0 - tr)) \wedge (tr' = tr_0 \wedge ref' = ref_0)$

(prop. calculus)

$= wait' = wait_0 \wedge ok' = ok_0 \wedge tr' = tr_0 \wedge ref' = ref_0$ (def. of $\mathbb{I}_{\text{rel}}$)

$= \mathbb{I}_{\text{rel}}[v_0/v]$

Lemma 4.46 (idempotent-$+^{\delta}$).

$$P +^{\delta} P = P$$

Proof of Lemma 4.46.

$$\begin{aligned}
& P +^{\delta} P = && \text{(def. of } +^{\delta}\text{)}\\
& = (P \curlywedge P); M_{+} && \text{(Lemma 4.22)}\\
& = \exists v_0, v_1 \bullet P[v_0/v'] \wedge P[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(rename variables)}\\
& = \exists v_0 \bullet P[v_0/v'] \wedge P[v_0/v'] \wedge M_{+}[v_0, v_0/0.v, 1.v] && \text{(prop. calculus)}\\
& = \exists v_0 \bullet P[v_0/v'] \wedge M_{+}[v_0, v_0/0.v, 1.v] && \text{(Lemma 4.45)}\\
& = \exists v_0 \bullet P[v_0/v'] \wedge \mathbb{I}_{\mathrm{rel}}[v_0, v] && \text{(def. of ;)}\\
& = P; \mathbb{I}_{\mathrm{rel}} && \text{(; unit)}\\
& = P
\end{aligned}$$

Lemma 4.47 (commutative-$+^{\delta}$).

$$P +^{\delta} Q = Q +^{\delta} P$$

Proof of Lemma 4.47.

$$\begin{aligned}
& P +^{\delta} Q = && \text{(def. of } +^{\delta}\text{)}\\
& = (P \curlywedge Q); M_{+} && \text{(Lemma 4.22)}\\
& = \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(rename variables)}\\
& = \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_1, v_0/0.v, 1.v] && \text{(Lemma 4.43)}\\
& = \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge (M_{+}[0.v, 1.v/1.v, 0.v])[v_1, v_0/0.v, 1.v] && \text{(def. of [])}\\
& = \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_1, v_0/1.v, 0.v] && \text{(def. of [])}\\
& = \exists v_0, v_1 \bullet P[v_1/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(prop. calculus)}\\
& = \exists v_0, v_1 \bullet Q[v_0/v'] \wedge P[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(Lemma 4.22)}\\
& = (Q \curlywedge P); M_{+} && \text{(def. of } +^{\delta}\text{)}\\
& = Q +^{\delta} P
\end{aligned}$$

Lemma 4.48 (closure-$+^{\delta}$-R1).

$$\mathbf{R1}(P\ +^{\delta}\ Q) = P\ +^{\delta}\ Q \text{ provided P and Q are } \mathbf{R1} \text{ healthy}$$

Proof of Lemma 4.48.

$$\begin{aligned}
&\mathbf{R1}(P\ +^{\delta}\ Q) = && \text{(assumption)}\\
&= \mathbf{R1}(\mathbf{R1}(P)\ +^{\delta}\ \mathbf{R1}(Q)) = && \text{(def. of } \mathbf{R1}\text{)}\\
&= ((P \wedge (tr \leq tr'))\ +^{\delta}\ (Q \wedge (tr \leq tr'))) \wedge (tr \leq tr') && \text{(def. of } \curlywedge\text{)}\\
&= (((P \wedge (tr \leq tr')) \curlywedge (Q \wedge (tr \leq tr'))); M_{+}) \wedge (tr \leq tr') && \text{(Lemma 4.22)}\\
&= (\exists v_0, v_1 \bullet (P \wedge (tr \leq tr'))[v_0/v'] \wedge (Q \wedge (tr \leq tr'))[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]) \wedge (tr \leq tr') && \text{(def. of [])}\\
&= (\exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \leq tr_0) \wedge Q[v_1/v'] \wedge (tr \leq tr_1) \wedge M_{+}[v_0, v_1/0.v, 1.v]) \wedge (tr \leq tr') && \text{(Lemma 4.40)}\\
&= (\exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \leq tr_0) \wedge Q[v_1/v'] \wedge (tr \leq tr_1) \wedge\\
&\qquad (M_{+} \wedge (tr' = 0.tr \vee tr' = 1.tr))[v_0, v_1/0.v, 1.v]) \wedge (tr \leq tr') && \text{(def. of [])}\\
&= (\exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \leq tr_0) \wedge Q[v_1/v'] \wedge (tr \leq tr_1) \wedge\\
&\qquad M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr' = tr_0 \vee tr' = tr_1)) \wedge (tr \leq tr') && \text{(prop. calculus)}\\
&= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (tr \leq tr_0) \wedge Q[v_1/v'] \wedge (tr \leq tr_1) \wedge\\
&\qquad M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr' = tr_0 \vee tr' = tr_1) \wedge (tr \leq tr') && \text{(prop. calculus)}\\
&= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr \leq tr_0) \wedge (tr \leq tr_1) \wedge (tr' = tr_0) \wedge (tr \leq tr') \vee\\
&\qquad P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr \leq tr_0) \wedge (tr \leq tr_1) \wedge (tr' = tr_1) \wedge (tr \leq tr')\\
& && \text{(properties of =)}\\
&= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr \leq tr_0) \wedge (tr \leq tr_1) \wedge (tr' = tr_0) \vee\\
&\qquad P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr \leq tr_0) \wedge (tr \leq tr_1) \wedge (tr' = tr_1) && \text{(prop. calculus)}\\
&= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (tr \leq tr_0) \wedge (tr \leq tr_1) \wedge ((tr' = tr_0) \vee (tr' = tr_1))\\
& && \text{(prop. calculus and def. of [])}\\
&= \exists v_0, v_1 \bullet (P \wedge (tr \leq tr'))[v_0/v'] \wedge (Q \wedge (tr \leq tr'))[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(def. of } \mathbf{R1}\text{)}\\
&= \exists v_0, v_1 \bullet (\mathbf{R1}(P))[v_0/v'] \wedge (\mathbf{R1}(Q))[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(assumption)}\\
&= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] && \text{(Lemma 4.22)}\\
&= (P \curlywedge Q); M_{+} && \text{(def. of } +^{\delta}\text{)}\\
&= P\ +^{\delta}\ Q
\end{aligned}$$

Lemma 4.49 (closure-$+^{\delta}$-R2).

$$\mathbf{R2}(P\ +^{\delta}\ Q) = P\ +^{\delta}\ Q \text{ provided P and Q are } \mathbf{R2} \text{ healthy}$$

Proof of Lemma 4.49.

$\mathbf{R2}(P(tr,tr')\ +^{\delta}\ Q(tr,tr')) =$ (def. of **R2**)

$= (P(tr,tr')\ +^{\delta}\ Q(tr,tr'))(\langle\rangle\, ,tr'-tr))$ (def. of $+^{\delta}$)

$= ((P(tr,tr') \curlywedge Q(tr,tr'));M_{+}(tr,tr'))(\langle\rangle\, ,tr'-tr))$ (Lemma 4.22)

$= (\exists v_0,v_1 \bullet P(tr,tr')[v_0/v'] \wedge Q(tr,tr')[v_1/v'] \wedge M_{+}[v_0,v_1/0.v,1.v](tr,tr'))(\langle\rangle\, ,tr'-tr))$ (assumption)

$= (\exists v_0,v_1 \bullet \mathbf{R2}(P(tr,tr'))[v_0/v'] \wedge \mathbf{R2}(Q)(tr,tr')[v_1/v'] \wedge M_{+}[v_0,v_1/0.v,1.v](tr,tr'))(\langle\rangle\, ,tr'-tr))$ (def. of **R2**)

$= (\exists v_0,v_1 \bullet P(\langle\rangle\, ,tr'-tr)[v_0/v'] \wedge Q(\langle\rangle\, ,tr'-tr)[v_1/v'] \wedge M_{+}[v_0,v_1/0.v,1.v](tr,tr'))(\langle\rangle\, ,tr'-tr)$ (substitution)

$= (\exists v_0,v_1 \bullet P[v_0/v'](\langle\rangle\, ,tr_0-tr) \wedge Q[v_1/v'](\langle\rangle\, ,tr_1-tr) \wedge M_{+}[v_0,v_1/0.v,1.v](tr,tr'))(\langle\rangle\, ,tr'-tr)$

(def. of M_{+} and substitution)

$$\begin{aligned}
= {} & \exists v_0,v_1 \bullet P[v_0/v'](\langle\rangle\, ,tr_0-tr) \wedge Q[v_1/v'](\langle\rangle\, ,tr_1-tr) \wedge \\
& \left(\left(\begin{array}{l}
((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\
\neg initQuiet(tr_0-tr) \vee \neg initQuiet(tr_1-tr) \Rightarrow \neg initQuiet(tr'-tr)) \wedge \\
((tr' = tr_0 \wedge ref' = \dots) \vee (tr' = tr_1 \wedge ref' = dots))
\end{array}\right) \lhd \mathbb{\delta}^{\delta} \rhd \right. \\
& \quad \left. \left(\begin{array}{l}
((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\
\neg initQuiet(tr_0-tr) \vee \neg initQuiet(tr_1-tr) \Rightarrow \neg initQuiet(tr'-tr)) \wedge \\
((tr' = tr_0 \wedge ref' = \dots) \vee (tr' = tr_1 \wedge ref' = dots))
\end{array}\right)\right)(\langle\rangle\, ,tr'-tr)
\end{aligned}$$

(substitution)

$$\begin{aligned}
= {} & \exists v_0,v_1 \bullet P[v_0/v'](\langle\rangle\, ,tr_0-tr) \wedge Q[v_1/v'](\langle\rangle\, ,tr_1-tr) \wedge \\
& \left(\left(\begin{array}{l}
((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\
\neg initQuiet((tr_0-tr)-\langle\rangle) \vee \neg initQuiet((tr_1-tr)-\langle\rangle) \Rightarrow \neg initQuiet((tr'-tr)-\langle\rangle)) \wedge \\
((tr'-tr = tr_0-tr \wedge ref' = \dots) \vee (tr'-tr = tr_1-tr \wedge ref' = dots))
\end{array}\right) \lhd \mathbb{\delta}^{\delta} \rhd \right. \\
& \quad \left. \left(\begin{array}{l}
((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\
\neg initQuiet((tr_0-tr)-\langle\rangle) \vee \neg initQuiet((tr_1-tr)-\langle\rangle) \Rightarrow \neg initQuiet((tr'-tr)-\langle\rangle)) \wedge \\
((tr'-tr = tr_0-tr \wedge ref' = \dots) \vee (tr'-tr = tr_1-tr \wedge ref' = dots))
\end{array}\right)\right)
\end{aligned}$$

(prop. calculus)

$$\begin{aligned}
= {} & \exists v_0,v_1 \bullet P[v_0/v'](\langle\rangle\, ,tr_0-tr) \wedge Q[v_1/v'](\langle\rangle\, ,tr_1-tr) \wedge \\
& \left(\left(\begin{array}{l}
((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\
\neg initQuiet(tr_0-tr) \vee \neg initQuiet(tr_1-tr) \Rightarrow \neg initQuiet(tr'-tr)) \wedge \\
((tr' = tr_0 \wedge ref' = \dots) \vee (tr' = tr_1 \wedge ref' = dots))
\end{array}\right) \lhd \mathbb{\delta}^{\delta} \rhd \right. \\
& \quad \left. \left(\begin{array}{l}
((wait_0 \iff wait_1) \wedge wait' = wait_0 \wedge (ok' = (ok_0 \wedge ok_1)) \wedge \\
\neg initQuiet(tr_0-tr) \vee \neg initQuiet(tr_1-tr) \Rightarrow \neg initQuiet(tr'-tr)) \wedge \\
((tr' = tr_0 \wedge ref' = \dots) \vee (tr' = tr_1 \wedge ref' = dots))
\end{array}\right)\right)
\end{aligned}$$

(def. of M_{+} and substitution)

$= \exists v_0,v_1 \bullet P[v_0/v'](\langle\rangle\, ,tr_0-tr) \wedge Q[v_1/v'](\langle\rangle\, ,tr_1-tr) \wedge M_{+}[v_0,v_1/0.v,1.v]$ (substitution)

$= \exists v_0,v_1 \bullet (P(\langle\rangle\, ,tr'-tr))[v_0/v'] \wedge (Q(\langle\rangle\, ,tr'-tr))[v_1/v'] \wedge M_{+}[v_0,v_1/0.v,1.v]$ (def. of **R2**)

$= \exists v_0,v_1 \bullet \mathbf{R2}(P(tr,tr'))[v_0/v'] \wedge \mathbf{R2}(Q(tr,tr'))[v_1/v'] \wedge M_{+}[v_0,v_1/0.v,1.v]$ (assumption)

$= \exists v_0,v_1 \bullet P(tr,tr')[v_0/v'] \wedge Q(tr,tr')[v_1/v'] \wedge M_{+}[v_0,v_1/0.v,1.v]$ (Lemma 4.22)

$= (P(tr,tr') \curlywedge Q(tr,tr'));M_{+}[v_0,v_1/0.v,1.v]$ (Lemma 4.22)

$= (P(tr,tr') \curlywedge Q(tr,tr'));M_{+}[v_0,v_1/0.v,1.v]$ (def. of $+^{\delta}$)

$= P(tr,tr')\ +^{\delta}\ Q(tr,tr')$

Lemma 4.50 (closure-$+^\delta$-$\mathbf{R3}^\delta$).

$$\mathbf{R3}^\delta(P \ +^\delta \ Q) = P \ +^\delta \ Q \text{ provided P and Q are } \mathbf{R3}^\delta \text{ healthy}$$

Proof of Lemma 4.50.

$\mathbf{R3}^\delta(P \ +^\delta \ Q) =$ (def. of $\mathbf{R3}^\delta$)

$= \mathbb{I}^\delta \lhd wait \rhd (P \ +^\delta \ Q)$ (Lemma 4.46)

$= (\mathbb{I}^\delta \ +^\delta \ \mathbb{I}^\delta) \lhd wait \rhd (P \ +^\delta \ Q)$ (def. of $+^\delta$)

$= (\mathbb{I}^\delta \curlywedge \mathbb{I}^\delta); M_+ \lhd wait \rhd (P \curlywedge Q); M_+$ (Lemma 4.22)

$= (\exists v_0, v_1 \bullet \mathbb{I}^\delta[v_0/v'] \land \mathbb{I}^\delta[v_1/v'] \land M_+[v_0, v_1/0.v, 1.v]) \lhd wait \rhd$
$\quad (\exists v_0, v_1 \bullet P[v_0/v'] \land Q[v_1/v'] \land M_+[v_0, v_1/0.v, 1.v])$ (wait not quantified)

$= \exists v_0, v_1 \bullet ((\mathbb{I}^\delta[v_0/v'] \land \mathbb{I}^\delta[v_1/v'] \land M_+[v_0, v_1/0.v, 1.v]) \lhd wait \rhd$
$\quad (P[v_0/v'] \land Q[v_1/v'] \land M_+[v_0, v_1/0.v, 1.v]))$ ($\land$-if distr.)

$= \exists v_0, v_1 \bullet ((\mathbb{I}^\delta[v_0/v'] \land \mathbb{I}^\delta[v_1/v']) \lhd wait \rhd$
$\quad (P[v_0/v'] \land Q[v_1/v'])) \land M_+[v_0, v_1/0.v, 1.v]$ (mutual distribution)

$= \exists v_0, v_1 \bullet (\mathbb{I}^\delta[v_0/v'] \lhd wait \rhd P[v_0/v']) \land (\mathbb{I}^\delta[v_1/v'] \lhd wait \rhd Q[v_1/v']) \land$
$\quad M_+[v_0, v_1/0.v, 1.v]$ (distr. of [])

$= \exists v_0, v_1 \bullet (\mathbb{I}^\delta \lhd wait \rhd P)[v_0/v'] \land (\mathbb{I}^\delta \lhd wait \rhd Q)[v_1/v'] \land$
$\quad M_+[v_0, v_1/0.v, 1.v]$ (def. of $\mathbf{R3}^\delta$)

$= \exists v_0, v_1 \bullet \mathbf{R3}^\delta P[v_0/v'] \land \mathbf{R3}^\delta(Q)[v_1/v'] M_+[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \land Q[v_1/v'] M_+[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q); M_+$ (def. of $+^\delta$)

$= P \ +^\delta \ Q$

Lemma 4.51 (closure-$+^{\delta}$-IOCO1).

$$\mathbf{IOCO1}(P\ +^{\delta}\ Q) = P\ +^{\delta}\ Q \text{ provided P and Q are } \mathbf{IOCO1} \text{ healthy}$$

Proof of Lemma 4.51.

$$\begin{array}{lr}
\mathbf{IOCO1}(P\ +^{\delta}\ Q) = & \text{(def. } +^{\delta}\text{)} \\
= \mathbf{IOCO1}(P \Yup Q; M_{+}) & \text{(Lemma 4.22)} \\
= \mathbf{IOCO1}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]) & \text{(def. } \mathbf{IOCO1}\text{)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (ok \Rightarrow (wait' \vee ok')) & \text{(Lemma 4.41)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge \left(\begin{array}{l} (0.wait \iff 1.wait) \wedge \\ (wait' = (0.wait \vee 1.wait)) \wedge \\ (ok' = (0.ok \wedge 1.ok)) \wedge M_{+} \end{array}\right) [v_0, v_1/0.v, 1.v] \wedge (ok \Rightarrow (wait' \vee ok')) & \\
 & \text{(def. of [])} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait_0 \iff wait_1) \wedge & \\
\qquad (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge (ok \Rightarrow (wait' \vee ok')) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge & \\
\qquad (ok \Rightarrow (wait_0 \vee wait_1 \vee (ok_0 \wedge ok_1))) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0 \vee wait_1)) \wedge (ok \Rightarrow (ok_1 \vee wait_0 \vee wait_1)) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (wait_0 \iff wait_1) \wedge (wait' = (wait_0 \vee wait_1)) \wedge (ok' = (ok_0 \wedge ok_1)) \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0)) \wedge (ok \Rightarrow (ok_1 \vee wait_1)) & \text{(prop. calculus)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge & \\
\qquad ((0.wait \iff 1.wait) \wedge (wait' = (0.wait \vee 1.wait)) \wedge (ok' = (0.ok \wedge 1.ok)) \wedge M_{+})[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0)) \wedge (ok \Rightarrow (ok_1 \vee wait_1)) & \text{(Lemma 4.41)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge & \\
\qquad (ok \Rightarrow (ok_0 \vee wait_0)) \wedge (ok \Rightarrow (ok_1 \vee wait_1)) & \text{(def. of [])} \\
= \exists v_0, v_1 \bullet (P \wedge (ok \Rightarrow (ok' \vee wait')))[v_0/v'] \wedge & \\
\qquad (Q \wedge (ok \Rightarrow (ok' \vee wait')))[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] & \text{(def. } \mathbf{IOCO1}\text{)} \\
= \exists v_0, v_1 \bullet \mathbf{IOCO1}(P)[v_0/v'] \wedge \mathbf{IOCO1}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] & \text{(assumption)} \\
= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] & \text{(Lemma 4.22)} \\
= P \Yup Q; M_{+} & \text{(def. of } +^{\delta}\text{)} \\
= P\ +^{\delta}\ Q &
\end{array}$$

Lemma 4.52 (closure-$+^{\delta}$-IOCO2).

$$\mathbf{IOCO2}(P\ +^{\delta}\ Q) = P\ +^{\delta}\ Q \text{ provided P and Q are } \mathbf{IOCO2} \text{ healthy}$$

Proof of Lemma 4.52.

$\mathbf{IOCO2}(P\ +^{\delta}\ Q) =$ (def. of $+^{\delta}$)

$= \mathbf{IOCO2}((P \curlywedge Q);M_+)$ (Lemma 4.22)

$= \mathbf{IOCO2}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_+[v_0, v_1/0.v, 1.v])$ (def. of **IOCO2**)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_+[v_0, v_1/0.v, 1.v] \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$

(assumption)

$= \exists v_0, v_1 \bullet \mathbf{IOCO2}(P)[v_0/v'] \wedge \mathbf{IOCO2}(Q)[v_1/v'] \wedge M_+[v_0, v_1/0.v, 1.v] \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. **IOCO2** and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge Q[v_1/v'] \wedge$
$\quad (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_+[v_0, v_1/0.v, 1.v] \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow (\neg wait' \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref'))))$ (Lemma 4.43 and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge Q[v_1/v'] \wedge$
$\quad (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_+[v_0, v_1/0.v, 1.v] \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow (\neg wait' \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref')))) \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$

$$\left(\left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{array}\right) \lhd \delta^{\delta} \rhd \left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{array}\right)\right)$$

(prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge Q[v_1/v'] \wedge$
$\quad (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_+[v_0, v_1/0.v, 1.v] \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$

$$\left(\left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{array}\right) \lhd \delta^{\delta} \rhd \left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{array}\right)\right)$$

(substitution and Lemma 4.43)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow (\neg wait_0 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_0)))) \wedge$
$\qquad Q[v_1/v'] \wedge (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow (\neg wait_1 \vee \forall o \in \mathcal{A}_{out} \bullet o \in ref_1)))) \wedge M_+[v_0, v_1/0.v, 1.v]$

(substitution and def. of **IOCO2**)

$= \exists v_0, v_1 \bullet \mathbf{IOCO2}(P)[v_0/v'] \wedge \mathbf{IOCO2}(Q)[v_1/v'] \wedge M_+[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_+[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q);M_+$ (def. of $+^{\delta}$)

$= P\ +^{\delta}\ Q$

Lemma 4.53 (closure-$+^{\delta}$-IOCO3).

$$\mathbf{IOCO3}(P\ +^{\delta}\ Q) = P\ +^{\delta}\ Q \text{ provided P and Q are } \mathbf{IOCO3} \text{ healthy}$$

Proof of Lemma 4.53.

$\mathbf{IOCO3}(P\ +^{\delta}\ Q) =$ (def. of $+^{\delta}$)

$= \mathbf{IOCO3}((P \curlywedge Q);M_{+})$ (Lemma 4.22)

$= \mathbf{IOCO3}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v])$ (def. of **IOCO3**)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\frown\delta^*)))$

(assumption)

$= \exists v_0, v_1 \bullet \mathbf{IOCO3}(P)[v_0/v'] \wedge \mathbf{IOCO3}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$(\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\frown\delta^*)))$ (def. **IOCO3** and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s\frown\delta^*))) \wedge Q[v_1/v'] \wedge$
$(\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s\frown\delta^*))) \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$(\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\frown\delta^*)))$ (Lemma 4.43 and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s\frown\delta^*))) \wedge Q[v_1/v'] \wedge$
$(\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s\ \delta^*))) \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$(\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s\frown\delta^*))) \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$

$$\left(\left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{array}\right) \lhd \mathbb{\delta}^{\delta} \rhd \left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{array}\right)\right)$$

(prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s\frown\delta^*))) \wedge Q[v_1/v'] \wedge$
$(\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s\frown\delta^*))) \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$

$$\left(\left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{array}\right) \lhd \mathbb{\delta}^{\delta} \rhd \left(\begin{array}{l} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{array}\right)\right)$$

(substitution and Lemma 4.43)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (\neg wait \Rightarrow (wait_0 \Rightarrow (\delta \notin ref_0 \Rightarrow \exists s \bullet tr_0 - tr \in s\frown\delta^*))) \wedge$
$Q[v_1/v'] \wedge (\neg wait \Rightarrow (wait_1 \Rightarrow (\delta \notin ref_1 \Rightarrow \exists s \bullet tr_1 - tr \in s\frown\delta^*))) \wedge M_{+}[v_0, v_1/0.v, 1.v]$

(substitution and def. of **IOCO3**)

$= \exists v_0, v_1 \bullet \mathbf{IOCO3}(P)[v_0/v'] \wedge \mathbf{IOCO3}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q);M_{+}$ (def. of $+^{\delta}$)

$= P\ +^{\delta}\ Q$

A.1.2 Section 4.3 (IOCO Implementations)

Lemma 4.54 (IOCO4-idempotent).

$$\mathbf{IOCO4} \circ \mathbf{IOCO4} = \mathbf{IOCO4}$$

Proof of Lemma 4.54.

$\mathbf{IOCO4}(\mathbf{IOCO4}(P)) =$ (def. of **IOCO4**)

$= \mathbf{IOCO4}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (def. of **IOCO4**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (def. of **IOCO4**)

$= \mathbf{IOCO4}(P)$

Lemma 4.55 (commutativity-IOCO4-R1).

$$\mathbf{IOCO4} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{IOCO4}$$

Proof of Lemma 4.55.

$\mathbf{IOCO4}(\mathbf{R1}(P)) =$ (def. of **R1**)

$= \mathbf{IOCO4}(P \wedge (tr \leq tr'))$ (def. of **IOCO4**)

$= P \wedge (tr \leq tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \wedge (tr \leq tr')$ (def. of **IOCO4**)

$= \mathbf{IOCO4}(P) \wedge (tr \leq tr')$ (def. of **R1**)

$= \mathbf{R1}(\mathbf{IOCO4}(P))$

Lemma 4.56 (commutativity-IOCO4-R2).

$$\mathbf{IOCO4} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{IOCO4}$$

Proof of Lemma 4.56.

$\mathbf{IOCO4}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)

$= \mathbf{IOCO4}(P(\langle\rangle, tr' - tr))$ (def. of **IOCO4**)

$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (tr',tr are not used in **IOCO4**)

$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)))(\langle\rangle, tr' - tr)$ (def. of **IOCO4**)

$= \mathbf{IOCO4}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)

$= \mathbf{R2}(\mathbf{IOCO4}(P))$

Lemma 4.57 (commutativity-IOCO4-R3).

$$\mathbf{IOCO4} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{IOCO4}$$

Proof of Lemma 4.57.

$\mathbf{IOCO4}(\mathbf{R3}_\iota^\delta(P)) =$ (def. of $\mathbf{R3}_\iota^\delta$)

$= \mathbf{IOCO4}(\mathbb{I}_\iota^\delta \lhd wait \rhd P)$ (def. of $\mathbf{IOCO4}$)

$= (\mathbb{I}_\iota^\delta \lhd wait \rhd P) \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ ($\land$-if-distr)

$= (\mathbb{I}_\iota^\delta \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))) \lhd wait \rhd$
$\quad (P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)))$ (def. of if and $\neg wait$)

$= \mathbb{I}_\iota^\delta \lhd wait \rhd (P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)))$ (def. of $\mathbf{IOCO4}$)

$= \mathbb{I}_\iota^\delta \lhd wait \rhd (\mathbf{IOCO4}(P))$ (def. of $\mathbf{R3}_\iota^\delta$)

$= \mathbf{R3}_\iota^\delta(\mathbf{IOCO4}(P))$

Lemma 4.58 (commutativity-IOCO4-IOCO1).

$$\mathbf{IOCO4} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{IOCO4}$$

Proof of Lemma 4.58.

$\mathbf{IOCO4}(\mathbf{IOCO1}(P)) =$ (def. of $\mathbf{IOCO1}$)

$= \mathbf{IOCO4}(P \land (ok \Rightarrow (wait' \lor ok')))$ (def. of $\mathbf{IOCO4}$)

$= P \land (ok \Rightarrow (wait' \lor ok')) \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (prop. calculus)

$= P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \land (ok \Rightarrow (wait' \lor ok'))$ (def. of $\mathbf{IOCO4}$)

$= \mathbf{IOCO4}(P) \land (ok \Rightarrow (wait' \lor ok'))$ (def. of $\mathbf{IOCO1}$)

$= \mathbf{IOCO1}(\mathbf{IOCO4}(P))$

Lemma 4.59 (commutativity-IOCO4-IOCO2).

$$\mathbf{IOCO4} \circ \mathbf{IOCO2} = \mathbf{IOCO2} \circ \mathbf{IOCO4}$$

Proof of Lemma 4.59.

$\mathbf{IOCO4}(\mathbf{IOCO2}(P)) =$ (def. of $\mathbf{IOCO2}$)

$= \mathbf{IOCO4}(P \land (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))))$ (def. of $\mathbf{IOCO4}$)

$= P \land (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \land$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (prop. calculus)

$= P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \land$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of $\mathbf{IOCO4}$)

$= \mathbf{IOCO4}(P) \land (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of $\mathbf{IOCO2}$)

$= \mathbf{IOCO2}(\mathbf{IOCO4}(P))$

Lemma 4.60 (commutativity-IOCO4-IOCO3).

$$\mathbf{IOCO4} \circ \mathbf{IOCO3} = \mathbf{IOCO3} \circ \mathbf{IOCO4}$$

Proof of Lemma 4.60.

$\mathbf{IOCO4}(\mathbf{IOCO3}(P)) =$ (def. of **IOCO3**)

$= \mathbf{IOCO4}(P \land (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*))))$ (def. of **IOCO4**)

$= P \land (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*))) \land$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$ (prop. calculus)

$= P \land (\neg wait \Rightarrow (wait' \Rightarrow (|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \land$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*)))$ (def. of **IOCO4**)

$= \mathbf{IOCO4}(P) \land (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s \frown \delta^*)))$ (def. of **IOCO3**)

$= \mathbf{IOCO3}(\mathbf{IOCO4}(P))$

Lemma 4.61 (closure-;-IOCO4).

$$\mathbf{IOCO4}(P;Q) = P;Q \text{ provided that P and Q are } \mathbf{IOCO4} \text{ healthy and Q is } \mathbf{R3}_\iota^\delta \text{ healthy}$$

Proof of Lemma 4.61.

$\mathbf{IOCO4}(P;Q) =$ (assumption and def. of $\mathbf{R3}_\iota^\delta$ and ;)

$= \mathbf{IOCO4}(\exists v_0 \bullet P[v_0/v'] \wedge (\mathbb{I}_\iota^\delta \lhd wait \rhd Q)[v_0/v])$ (def. **IOCO4**, if, and substitution)

$= \exists v_0 \bullet P[v_0/v'] \wedge (\mathbb{I}_\iota^\delta[v_0/v] \wedge wait_0 \vee \neg wait_0 \wedge Q[v_0/v]) \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|))$

(def. of $\mathbb{I}_\iota^\delta$ and prop. calculus)

$$= \exists v_0 \bullet \left(\left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \wedge \\ \left(\begin{array}{l} \neg ok \wedge \cdots \wedge (\neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \vee \\ ok' \wedge \cdots \wedge (wait' = wait) \wedge (ref'_{in} = ref_{in}) \end{array} \right) [v_0/v] \wedge wait_0 \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \end{array} \right) \wedge \left(\begin{array}{l} wait \vee \\ \neg wait' \vee \\ ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|) \end{array} \right) \right)$$

(substitution and prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge \\ (\neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right) \vee \\ \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \wedge \\ (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge (\neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \vee \\ \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \wedge \\ (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right)$$

(substitution)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge (\neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \vee \\ \left(\begin{array}{l} (P \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)))[v_0/v'] \wedge wait_0 \wedge \\ ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \end{array} \right) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right)$$

(def. of **IOCO4** and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \neg ok_0 \wedge \cdots \wedge (\neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \vee \\ P[v_0/v'] \wedge wait_0 \wedge ok' \wedge \cdots \wedge (wait' = wait_0) \wedge (ref'_{in} = ref_{in0}) \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right)$$

(prop. calculus and def. of $\mathbb{I}_\iota^\delta$ and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}_\iota^\delta[v_0/v] \vee \\ \mathbf{IOCO4}(P)[v_0/v'] \wedge \neg wait_0 \wedge \mathbf{IOCO4}[v_0/v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right)$$

(def. **IOCO4** and renaming)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}_\iota^\delta[v_0/v] \vee \\ \left(\begin{array}{l} P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out0}|)) \wedge \neg wait_0 \wedge \\ Q[v_0/v] \wedge (wait_0 \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \wedge \\ (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}_\iota^\delta[v_0/v] \vee \left(\begin{array}{l} P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (\neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \wedge \\ (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right) \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}_\iota^\delta[v_0/v] \vee \\ P[v_0/v'] \wedge \neg wait_0 \wedge Q[v_0/v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}|-1) \leq |ref'_{out}|)) \end{array} \right)$$

(substitution and def. of **IOCO4**)

$= \exists v_0 \bullet P[v_0/v'] \wedge wait_0 \wedge \mathbb{I}_\iota^\delta[v_0/v] \vee P[v_0/v'] \wedge \neg wait_0 \wedge \mathbf{IOCO4}(Q)[v_0/v]$ (assumption and def. of $\mathbf{R3}_\iota^\delta$)

$= \exists v_0 \bullet P[v_0/v'] \wedge \mathbf{R3}_\iota^\delta(Q)[v_0/v]$ (assumption and def. of ;)

$= P;Q$

Lemma 4.62 (closure-$\sqcap^{\delta}$-IOCO4).

$$\mathbf{IOCO4}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided that P and Q are } \mathbf{IOCO4} \text{ healthy}$$

Proof of Lemma 4.62.

$$\mathbf{IOCO4}(P \sqcap^{\delta} Q) = \qquad \text{(def. of } \sqcap^{\delta}\text{)}$$

$$= \mathbf{IOCO4}((P \curlywedge Q);M_{\sqcap}) \qquad \text{(Lemma 4.22)}$$

$$= \mathbf{IOCO4}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]) \qquad \text{(def. of } \mathbf{IOCO4}\text{)}$$

$$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \qquad \text{(assumption)}$$

$$= \exists v_0, v_1 \bullet \mathbf{IOCO4}(P)[v_0/v'] \wedge \mathbf{IOCO4}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$$
$$\text{(def. } \mathbf{IOCO4} \text{ and substitution)}$$

$$\begin{aligned}= \exists v_0, v_1 \bullet\ & P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge Q[v_1/v'] \wedge \\ & (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))\end{aligned}$$
$$\text{(Lemma 4.29 and substitution)}$$

$$\begin{aligned}= \exists v_0, v_1 \bullet\ & P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge \\ & (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge \\ & (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \wedge (wait' = (wait_0 \wedge wait_1)) \wedge \\ & \left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \mathfrak{S}^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix} \right)\end{aligned}$$
$$\text{(prop. calculus)}$$

$$\begin{aligned}= \exists v_0, v_1 \bullet\ & P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge \\ & (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge \\ & (wait' = (wait_0 \wedge wait_1)) \wedge \\ & \left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \mathfrak{S}^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix} \right)\end{aligned}$$
$$\text{(substitution and Lemma 4.29)}$$

$$\begin{aligned}= \exists v_0, v_1 \bullet\ & P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge \\ & Q[v_1/v'] \wedge (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]\end{aligned}$$
$$\text{(substitution and def. of } \mathbf{IOCO4}\text{)}$$

$$= \exists v_0, v_1 \bullet \mathbf{IOCO4}(P)[v_0/v'] \wedge \mathbf{IOCO4}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \qquad \text{(assumption)}$$

$$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \qquad \text{(Lemma 4.22)}$$

$$= (P \curlywedge Q);M_{\sqcap} \qquad \text{(def. of } \sqcap^{\delta}\text{)}$$

$$= P \sqcap^{\delta} Q$$

Lemma 4.63 (closure-$+^{\delta}$-IOCO4).

$$\mathbf{IOCO4}(P\ +^{\delta}\ Q) = P\ \sqcap^{\delta}\ Q \text{ provided that P and Q are } \mathbf{IOCO4} \text{ healthy}$$

Proof of Lemma 4.63.

$\mathbf{IOCO4}(P\ +^{\delta}\ Q) =$ (def. of $+^{\delta}$)

$= \mathbf{IOCO4}((P \curlywedge Q);M_{+})$ (Lemma 4.22)

$= \mathbf{IOCO4}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v])$ (def. of **IOCO4**)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$

(assumption)

$= \exists v_0, v_1 \bullet \mathbf{IOCO4}(P)[v_0/v'] \wedge \mathbf{IOCO4}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$

(def. **IOCO4** and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge Q[v_1/v'] \wedge$
$(wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|))$

(Lemma 4.43 and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$(wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge$
$(wait \vee \neg wait' \vee ((|\mathcal{A}_{out}| - 1) \leq |ref'_{out}|)) \wedge (wait' = (wait_0 \wedge wait_1)) \wedge$
$\left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \mathfrak{S}^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$ (prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$(wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge$
$(wait' = (wait_0 \wedge wait_1)) \wedge$
$\left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \mathfrak{S}^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$

(substitution and Lemma 4.43)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,0}|)) \wedge$
$Q[v_1/v'] \wedge (wait \vee \neg wait_1 \vee ((|\mathcal{A}_{out}| - 1) \leq |ref_{out\,1}|)) \wedge M_{+}[v_0, v_1/0.v, 1.v]$

(substitution and def. of **IOCO4**)

$= \exists v_0, v_1 \bullet \mathbf{IOCO4}(P)[v_0/v'] \wedge \mathbf{IOCO4}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q);M_{+}$ (def. of $+^{\delta}$)

$= P\ +^{\delta}\ Q$

Lemma 4.64 (IOCO5-idempotent).

$$\mathbf{IOCO5} \circ \mathbf{IOCO5} = \mathbf{IOCO5}$$

Proof of Lemma 4.64.

$\mathbf{IOCO5}(\mathbf{IOCO5}(P)) =$ (def. of **IOCO5**)

$= \mathbf{IOCO5}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (def. of **IOCO5**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (def. of **IOCO5**)

$= \mathbf{IOCO5}(P)$

Lemma 4.65 (commutativity-IOCO5-R1).

$$\mathbf{IOCO5} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{IOCO5}$$

Proof of Lemma 4.65.

$\mathbf{IOCO5}(\mathbf{R1}(P)) =$ (def. of **R1**)

$= \mathbf{IOCO5}(P \wedge (tr \leq tr'))$ (def. of **IOCO5**)

$= P \wedge (tr \leq tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))) \wedge (tr \leq tr')$ (def. of **IOCO5**)

$= \mathbf{IOCO5}(P) \wedge (tr \leq tr')$ (def. of **R1**)

$= \mathbf{R1}(\mathbf{IOCO5}(P))$

Lemma 4.66 (commutativity-IOCO5-R2).

$$\mathbf{IOCO5} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{IOCO5}$$

Proof of Lemma 4.66.

$\mathbf{IOCO5}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)

$= \mathbf{IOCO5}(P(\langle\rangle, tr' - tr))$ (def. of **IOCO5**)

$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (tr',tr are not used in **IOCO5**)

$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))))(\langle\rangle, tr' - tr)$ (def. of **IOCO5**)

$= \mathbf{IOCO5}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)

$= \mathbf{R2}(\mathbf{IOCO5}(P))$

Lemma 4.67 (commutativity-IOCO5-R3).

$$\mathbf{IOCO5} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{IOCO5}$$

Proof of Lemma 4.67.

$\mathbf{IOCO5}(\mathbf{R3}^{\delta}_{\iota}(P)) =$ (def. of $\mathbf{R3}^{\delta}_{\iota}$)

$= \mathbf{IOCO5}(\mathbb{I}^{\delta}_{\iota} \lhd wait \rhd P)$ (def. of **IOCO5**)

$= (\mathbb{I}^{\delta}_{\iota} \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ ($\wedge$-if-distr)

$= (\mathbb{I}^{\delta}_{\iota} \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))) \lhd wait \rhd$
$\quad (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))))$ (def. of if and $\neg wait$)

$= \mathbb{I}^{\delta}_{\iota} \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))))$ (def. of **IOCO5**)

$= \mathbb{I}^{\delta}_{\iota} \lhd wait \rhd (\mathbf{IOCO5}(P))$ (def. of $\mathbf{R3}^{\delta}_{\iota}$)

$= \mathbf{R3}^{\delta}_{\iota}(\mathbf{IOCO5}(P))$

Lemma 4.68 (commutativity-IOCO5-IOCO1).

$$\mathbf{IOCO5} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{IOCO5}$$

Proof of Lemma 4.68.

$\mathbf{IOCO5}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{IOCO5}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **IOCO5**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO5**)
$= \mathbf{IOCO5}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{IOCO5}(P))$

Lemma 4.69 (commutativity-IOCO5-IOCO2).

$$\mathbf{IOCO5} \circ \mathbf{IOCO2} = \mathbf{IOCO2} \circ \mathbf{IOCO5}$$

Proof of Lemma 4.69.

$\mathbf{IOCO5}(\mathbf{IOCO2}(P)) =$ (def. of **IOCO2**)
$= \mathbf{IOCO5}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))))$ (def. of **IOCO5**)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of **IOCO5**)
$= \mathbf{IOCO5}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow quiescence)))$ (def. of **IOCO2**)
$= \mathbf{IOCO2}(\mathbf{IOCO5}(P))$

Lemma 4.70 (commutativity-IOCO5-IOCO3).

$$\mathbf{IOCO5} \circ \mathbf{IOCO3} = \mathbf{IOCO3} \circ \mathbf{IOCO5}$$

Proof of Lemma 4.70.

$\mathbf{IOCO5}(\mathbf{IOCO3}(P)) =$ (def. of **IOCO3**)
$= \mathbf{IOCO5}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s ^\frown \delta^*))))$ (def. of **IOCO5**)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s ^\frown \delta^*))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \emptyset))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s ^\frown \delta^*)))$ (def. of **IOCO5**)
$= \mathbf{IOCO5}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (\delta \notin ref' \Rightarrow \exists s \bullet tr' - tr = s ^\frown \delta^*)))$ (def. of **IOCO3**)
$= \mathbf{IOCO3}(\mathbf{IOCO5}(P))$

Lemma 4.71 (closure-;-IOCO5).

$\mathbf{IOCO5}(P;Q) = P;Q$ provided that P and Q are **IOCO5** healthy and Q is $\mathbf{R3}_\iota^\delta$ healthy

Proof of Lemma 4.71.

$\mathbf{IOCO5}(P;Q) =$ (assumption and def. of $\mathbf{R3}_\iota^\delta$ and ;)

$= \mathbf{IOCO5}(\exists v_0 \bullet P[v_0/v'] \land (\mathbb{I}_\iota^\delta \lhd wait \rhd Q)[v_0/v])$ (def. **IOCO5**, if, and substitution)

$= \exists v_0 \bullet P[v_0/v'] \land (\mathbb{I}_\iota^\delta[v_0/v] \land wait_0 \lor \neg wait_0 \land Q[v_0/v]) \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset))$

(def. of $\mathbb{I}_\iota^\delta$ and prop. calculus)

$$= \exists v_0 \bullet \left(\left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \land \\ \left(\begin{array}{l} \neg ok \land \cdots \land (\neg wait' \lor (ref'_{in} = \emptyset)) \lor \\ ok' \land \cdots \land (wait' = wait) \land (ref'_{in} = ref_{in}) \end{array} \right) [v_0/v] \land wait_0 \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \end{array} \right) \land \left(\begin{array}{l} wait \lor \\ \neg wait' \lor \\ (ref'_{in} = \emptyset) \end{array} \right) \right)$$

(substitution and prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land \\ (\neg wait' \lor (ref'_{in} = \emptyset)) \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right) \lor \\ \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \land \\ (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land (\neg wait' \lor (ref'_{in} = \emptyset)) \lor \\ \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \land \\ (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right)$$

(substitution)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land (\neg wait' \lor (ref'_{in} = \emptyset)) \lor \\ \left(\begin{array}{l} (P \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)))[v_0/v'] \land wait_0 \land \\ ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right)$$

(def. of **IOCO5** and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land (\neg wait' \lor (ref'_{in} = \emptyset)) \lor \\ P[v_0/v'] \land wait_0 \land ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right)$$

(prop. calculus and def. of $\mathbb{I}_\iota^\delta$ and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \mathbb{I}_\iota^\delta[v_0/v] \lor \\ \mathbf{IOCO5}(P)[v_0/v'] \land \neg wait_0 \land \mathbf{IOCO5}[v_0/v] \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right)$$

(def. **IOCO5** and renaming)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \mathbb{I}_\iota^\delta[v_0/v] \lor \\ \left(\begin{array}{l} P[v_0/v'] \land (wait \lor \neg wait_0 \lor (ref_{in0} = \emptyset)) \land \neg wait_0 \land \\ Q[v_0/v] \land (wait_0 \lor \neg wait' \lor (ref_{in}' = \emptyset)) \land \\ (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(P[v_0/v'] \land wait_0 \land \mathbb{I}_\iota^\delta[v_0/v] \lor \left(\begin{array}{l} P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (\neg wait' \lor (ref'_{in} = \emptyset)) \land \\ (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right) \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \mathbb{I}_\iota^\delta[v_0/v] \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor (ref'_{in} = \emptyset)) \end{array} \right)$$

(substitution and def. of **IOCO5**)

$= \exists v_0 \bullet P[v_0/v'] \land wait_0 \land \mathbb{I}_\iota^\delta[v_0/v] \lor P[v_0/v'] \land \neg wait_0 \land \mathbf{IOCO5}(Q)[v_0/v]$ (assumption and def. of $\mathbf{R3}_\iota^\delta$)

$= \exists v_0 \bullet P[v_0/v'] \land \mathbf{R3}_\iota^\delta(Q)[v_0/v]$ (assumption and def. of ;)

$= P;Q$

Lemma 4.72 (closure-$\sqcap^{\delta}$-IOCO5).

$$\mathbf{IOCO5}(P \sqcap^{\delta} Q) = P \sqcap^{\delta} Q \text{ provided that P and Q are } \mathbf{IOCO5} \text{ healthy}$$

Proof of Lemma 4.72.

$\mathbf{IOCO5}(P \sqcap^{\delta} Q) =$ (def. of $\sqcap^{\delta}$)

$= \mathbf{IOCO5}((P \curlywedge Q);M_{\sqcap})$ (Lemma 4.22)

$= \mathbf{IOCO5}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v])$ (def. of **IOCO5**)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset))$ (assumption)

$= \exists v_0, v_1 \bullet \mathbf{IOCO5}(P)[v_0/v'] \wedge \mathbf{IOCO5}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset))$

(def. **IOCO5** and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge Q[v_1/v'] \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge$
$\quad M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset))$ (Lemma 4.29 and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge$
$\quad (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset)) \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$
$$\left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \delta^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$$

(prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v] \wedge$
$\quad (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$
$$\left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \delta^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$$

(substitution and Lemma 4.29)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge$
$\quad Q[v_1/v'] \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$

(substitution and def. of **IOCO5**)

$= \exists v_0, v_1 \bullet \mathbf{IOCO5}(P)[v_0/v'] \wedge \mathbf{IOCO5}(Q)[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{\sqcap}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q);M_{\sqcap}$ (def. of $\sqcap^{\delta}$)

$= P \sqcap^{\delta} Q$

Lemma 4.73 (closure-$+^{\delta}$-IOCO5).

$$\mathbf{IOCO5}(P\ +^{\delta}\ Q) = P\ \sqcap^{\delta}\ Q \text{ provided that P and Q are } \mathbf{IOCO5} \text{ healthy}$$

Proof of Lemma 4.73.

$\mathbf{IOCO5}(P\ +^{\delta}\ Q) =$ (def. of $+^{\delta}$)

$= \mathbf{IOCO5}((P \curlywedge Q);M_{+})$ (Lemma 4.22)

$= \mathbf{IOCO5}(\exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v])$ (def. of **IOCO5**)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset))$ (assumption)

$= \exists v_0, v_1 \bullet \mathbf{IOCO5}(P)[v_0/v'] \wedge \mathbf{IOCO5}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset))$

(def. **IOCO5** and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge Q[v_1/v'] \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge$
$\quad M_{+}[v_0, v_1/0.v, 1.v] \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset))$

(Lemma 4.43 and substitution)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$\quad (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge (wait \vee \neg wait' \vee (ref'_{in} = \emptyset)) \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$
$$\left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \eth^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$$

(prop. calculus)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_0/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v] \wedge$
$\quad (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge$
$\quad (wait' = (wait_0 \wedge wait_1)) \wedge$
$$\left(\begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \ldots ref'_{in} = ref_{in1} \end{pmatrix} \lhd \eth^{\delta} \rhd \begin{pmatrix} tr' = tr_0 \wedge \cdots \wedge ref'_{in} = ref_{in0} \vee \\ tr' = tr_1 \wedge \cdots \wedge ref'_{in} = ref_{in1} \end{pmatrix}\right)$$

(substitution and Lemma 4.43)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge (wait \vee \neg wait_0 \vee (ref_{in0} = \emptyset)) \wedge$
$\quad Q[v_1/v'] \wedge (wait \vee \neg wait_1 \vee (ref_{in1} = \emptyset)) \wedge M_{+}[v_0, v_1/0.v, 1.v]$

(substitution and def. of **IOCO5**)

$= \exists v_0, v_1 \bullet \mathbf{IOCO5}(P)[v_0/v'] \wedge \mathbf{IOCO5}(Q)[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]$ (assumption)

$= \exists v_0, v_1 \bullet P[v_0/v'] \wedge Q[v_1/v'] \wedge M_{+}[v_0, v_1/0.v, 1.v]$ (Lemma 4.22)

$= (P \curlywedge Q);M_{+}$ (def. of $+^{\delta}$)

$= P\ +^{\delta}\ Q$

Lemma 4.74 ($\mathfrak{S}_\iota$-left-zero).

$$\mathfrak{S}_\iota ; P = \mathfrak{S}_\iota$$

Proof of Lemma 4.74.

$$\begin{aligned}
&\mathfrak{S}_\iota ; P = && \text{(closure of } \mathbf{R3}_\iota^\delta\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd (\mathfrak{S}_\iota ; P) && \text{(def. of } \mathfrak{S}_\iota\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((\mathbb{I}_\iota^\delta \lhd wait \rhd tr' - tr \in \mathcal{A}_{in}^* \land wait'); P) && \text{(def. of } \lhd_\rhd\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in \mathcal{A}_{in}^* \land wait'); P) && \text{(P meets } \mathbf{R3}\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in \mathcal{A}_{in}^* \land wait'); \mathbb{I}_\iota^\delta \lhd wait \rhd P) && \text{(P meets } \mathbf{R3}\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in \mathcal{A}_{in}^* \land wait'); \mathbb{I}_\iota^\delta \lhd wait \rhd P) && \text{(def. of ;)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in \mathcal{A}_{in}^* \land wait'); \mathbb{I}_\iota^\delta) && \text{(; unit)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in \mathcal{A}_{in}^* \land wait')) && \text{(def. of } \mathfrak{S}_\iota\text{)}\\
&= \mathfrak{S}_\iota
\end{aligned}$$

Lemma 4.75 ($\mathfrak{S}_\iota^\delta$-left-zero).

$$\mathfrak{S}_\iota^\delta ; P = \mathfrak{S}_\iota^\delta$$

Proof of Lemma 4.75.

$$\begin{aligned}
&\mathfrak{S}_\iota^\delta ; P = && \text{(closure of } \mathbf{R3}_\iota^\delta\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd (\mathfrak{S}_\iota^\delta ; P) && \text{(def. of } \mathfrak{S}_\iota^\delta\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((\mathbb{I}_\iota^\delta \lhd wait \rhd tr' - tr \in (\mathcal{A}_{in} \cup \delta)^* \land wait' \land wait'); P) && \text{(def. of } \lhd_\rhd\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in (\mathcal{A}_{in} \cup \delta)^* \land wait' \land wait'); P) && \text{(P meets } \mathbf{R3}\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in (\mathcal{A}_{in} \cup \delta)^* \land wait' \land wait'); \mathbb{I}_\iota^\delta \lhd wait \rhd P) && \text{(P meets } \mathbf{R3}\text{)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in (\mathcal{A}_{in} \cup \delta)^* \land wait' \land wait'); \mathbb{I}_\iota^\delta \lhd wait \rhd P) && \text{(def. of ;)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in (\mathcal{A}_{in} \cup \delta)^* \land wait' \land wait'); \mathbb{I}_\iota^\delta) && \text{(; unit)}\\
&= \mathbb{I}_\iota^\delta \lhd wait \rhd ((tr' - tr \in (\mathcal{A}_{in} \cup \delta)^* \land wait' \land wait')) && \text{(def. of } \mathfrak{S}_\iota^\delta\text{)}\\
&= \mathfrak{S}_\iota^\delta
\end{aligned}$$

Lemma 4.76 (Fair choice refinement).

$$P \sqcap^{\delta} Q \sqsubseteq P \sqcap_f^{\delta} Q$$

Proof of Lemma 4.76.

$P \sqcap_f^{\delta} Q =$ (definition of $\sqcap_f^{\delta}$)

$= \sqcap\{P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q | 0 < p < 1\}$ (definition of $\sqcap$)

$\sqsupseteq \sqcap\{P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q | 0 < p < 1\} \sqcap P \sqcap Q$ (laws for ${}_{p}\textcircled{$\sqcap$}^{\delta}$)

$= \sqcap\{P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q | 0 < p < 1\} \sqcap P\ {}_{1}\textcircled{$\sqcap$}^{\delta}\ Q \sqcap P\ {}_{0}\textcircled{$\sqcap$}^{\delta}\ Q$ (definition of $\sqcap$)

$= \sqcap\{P\ {}_{p}\textcircled{$\sqcap$}^{\delta}\ Q | 0 \leq p \leq 1\}$ (laws for ${}_{p}\textcircled{$\sqcap$}^{\delta}$)

$= P \sqcap^{\delta} Q$

A.1.3 Section 4.4 (Predicative Input-Output Conformance Relation)

Lemma 4.77 (not-ioco and refinement).

$$(((S \sqsubseteq_{ioco} I_2) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow S \sqsubseteq_{ioco} I_1) \ = \ (((S \not\sqsubseteq_{ioco} I_1) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow S \not\sqsubseteq_{ioco} I_2)$$

Proof of Lemma 4.77.

$((S \sqsubseteq_{ioco} I_2) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow (S \sqsubseteq_{ioco} I_1)$ (prop. calculus)

$= (S \not\sqsubseteq_{ioco} I_2) \vee (I_2 \not\sqsubseteq I_1) \vee (S \sqsubseteq_{ioco} I_1)$ (prop. calculus)

$= (S \sqsubseteq_{ioco} I_1) \vee (I_2 \not\sqsubseteq I_1) \vee (S \not\sqsubseteq_{ioco} I_2)$ (prop. calculus)

$= ((S \not\sqsubseteq_{ioco} I_1) \wedge (I_2 \sqsubseteq I_1)) \Rightarrow (S \not\sqsubseteq_{ioco} I_2)$

A.1.4 Section 4.5 (Test Cases, Test Processes, and Test Suites)

Lemma 4.78 (TC1-idempotent).

$$\mathbf{TC1} \circ \mathbf{TC1} = \mathbf{TC1}$$

Proof of Lemma 4.78.

$\mathbf{TC1}(\mathbf{TC1}(P)) =$ (def. of **TC1**)
$= \mathbf{TC1}(P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (prop. calculus)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= \mathbf{TC1}(P)$

Lemma 4.79 (commutativity-TC1-R1).

$$\mathbf{TC1} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC1}$$

Proof of Lemma 4.79.

$\mathbf{TC1}(\mathbf{R1}(P)) =$ (def. of **R1**)
$= \mathbf{TC1}(P \wedge (tr \leq tr'))$ (def. of **TC1**)
$= P \wedge (tr \leq tr') \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (prop. calculus)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (tr \leq tr')$ (def. of **TC1**)
$= \mathbf{TC1}(P) \wedge (tr \leq tr')$ (def. of **R1**)
$= \mathbf{R1}(\mathbf{TC1}(P))$

Lemma 4.80 (commutativity-TC1-R2).

$$\mathbf{TC1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC1}$$

Proof of Lemma 4.80.

$\mathbf{TC1}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)
$= \mathbf{TC1}(P(\langle\rangle, tr' - tr))$ (def. of **TC1**)
$= P(\langle\rangle, tr' - tr) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (tr',tr are not used in **TC1**)
$= (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))(\langle\rangle, tr' - tr)$ (def. of **TC1**)
$= \mathbf{TC1}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{TC1}(P))$

Lemma 4.81 (commutativity-TC1-IOCO1).

$$\mathbf{TC1} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC1}$$

Proof of Lemma 4.81.

$$\begin{aligned}
&\mathbf{TC1}(\mathbf{IOCO1}(P)) = && \text{(def. of } \mathbf{IOCO1}\text{)}\\
&= \mathbf{TC1}(P \wedge (ok \Rightarrow (wait' \vee ok'))) && \text{(def. of } \mathbf{TC1}\text{)}\\
&= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) && \text{(prop. calculus)}\\
&= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (ok \Rightarrow (wait' \vee ok')) && \text{(def. of } \mathbf{TC1}\text{)}\\
&= \mathbf{TC1}(P) \wedge (ok \Rightarrow (wait' \vee ok')) && \text{(def. of } \mathbf{IOCO1}\text{)}\\
&= \mathbf{IOCO1}(\mathbf{TC1}(P))
\end{aligned}$$

Lemma 4.82 (TC2-idempotent).

$$\mathbf{TC2} \circ \mathbf{TC2} = \mathbf{TC2}$$

Proof of Lemma 4.82.

$$\begin{aligned}
&\mathbf{TC2}(\mathbf{TC2}(P)) = && \text{(def. of } \mathbf{TC2}\text{)}\\
&= \mathbf{TC2}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) && \text{(def. of } \mathbf{TC2}\text{)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge \\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) && \text{(prop. calculus)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) && \text{(def. of } \mathbf{TC2}\text{)}\\
&= \mathbf{TC2}(P)
\end{aligned}$$

Lemma 4.83 (commutativity-TC2-R1).

$$\mathbf{TC2} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC2}$$

Proof of Lemma 4.83.

$$\begin{aligned}
&\mathbf{TC2}(\mathbf{R1}(P)) = && \text{(def. of } \mathbf{R1}\text{)}\\
&= \mathbf{TC2}(P \wedge (tr \leq tr')) && \text{(def. of } \mathbf{TC2}\text{)}\\
&= P \wedge (tr \leq tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) && \text{(prop. calculus)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge (tr \leq tr') && \text{(def. of } \mathbf{TC2}\text{)}\\
&= \mathbf{TC2}(P) \wedge (tr \leq tr') && \text{(def. of } \mathbf{R1}\text{)}\\
&= \mathbf{R1}(\mathbf{TC2}(P))
\end{aligned}$$

Lemma 4.84 (commutativity-TC2-R2).

$$\mathbf{TC2} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC2}$$

Proof of Lemma 4.84.

$\mathbf{TC2}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)
$= \mathbf{TC2}(P(\langle\rangle, tr' - tr))$ (def. of **TC2**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (tr',tr are not used in **TC2**)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))))(\langle\rangle, tr' - tr)$ (def. of **TC2**)
$= \mathbf{TC2}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{TC2}(P))$

Lemma 4.85 (commutativity-TC2-IOCO1).

$$\mathbf{TC2} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC2}$$

Proof of Lemma 4.85.

$\mathbf{TC2}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{TC2}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **TC2**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **TC2**)
$= \mathbf{TC2}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{TC2}(P))$

Lemma 4.86 (commutativity-TC2-TC1).

$$\mathbf{TC2} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC2}$$

Proof of Lemma 4.86.

$\mathbf{TC2}(\mathbf{TC1}(P)) =$ (def. of **TC1**)
$= \mathbf{TC2}(P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (def. of **TC2**)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (prop. calculus)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC2**)
$= \mathbf{TC2}(P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= \mathbf{TC1}(\mathbf{TC2}(P))$

Lemma 4.87 (TC3-idempotent).

$$\mathbf{TC3} \circ \mathbf{TC3} = \mathbf{TC3}$$

Proof of Lemma 4.87.

$\mathbf{TC3}(\mathbf{TC3}(P)) =$ (def. of **TC3**)
$= \mathbf{TC3}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (def. of **TC3**)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (def. of **TC3**)
$= \mathbf{TC3}(P)$

Lemma 4.88 (commutativity-TC3-R1).

$$\mathbf{TC3} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC3}$$

Proof of Lemma 4.88.

$\mathbf{TC3}(\mathbf{R1}(P)) =$ (def. of **R1**)
$= \mathbf{TC3}(P \wedge (tr \leq tr'))$ (def. of **TC3**)
$= P \wedge (tr \leq tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge (tr \leq tr')$ (def. of **TC3**)
$= \mathbf{TC3}(P) \wedge (tr \leq tr')$ (def. of **R1**)
$= \mathbf{R1}(\mathbf{TC3}(P))$

Lemma 4.89 (commutativity-TC3-R2).

$$\mathbf{TC3} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC3}$$

Proof of Lemma 4.89.

$\mathbf{TC3}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)
$= \mathbf{TC3}(P(\langle\rangle, tr' - tr))$ (def. of **TC3**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (tr',tr are not used in **TC3**)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))))(\langle\rangle, tr' - tr)$ (def. of **TC3**)
$= \mathbf{TC3}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{TC3}(P))$

Lemma 4.90 (commutativity-TC3-IOCO1).

$$\mathbf{TC3} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC3}$$

Proof of Lemma 4.90.

$\mathbf{TC3}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{TC3}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **TC3**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **TC3**)
$= \mathbf{TC3}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{TC3}(P))$

Lemma 4.91 (commutativity-TC3-TC1).

$$\mathbf{TC3} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC3}$$

Proof of Lemma 4.91.

$\mathbf{TC3}(\mathbf{TC1}(P)) =$ (def. of **TC1**)
$= \mathbf{TC3}(P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (def. of **TC3**)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC3**)
$= \mathbf{TC3}(P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= \mathbf{TC1}(\mathbf{TC3}(P))$

Lemma 4.92 (commutativity-TC3-TC2).

$$\mathbf{TC3} \circ \mathbf{TC2} = \mathbf{TC2} \circ \mathbf{TC3}$$

Proof of Lemma 4.92.

$\mathbf{TC3}(\mathbf{TC2}(P)) =$ (def. of **TC2**)
$= \mathbf{TC3}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))))$ (def. of **TC3**)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (def. of **TC3**)
$= \mathbf{TC3}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (def. of **TC2**)
$= \mathbf{TC2}(\mathbf{TC3}(P))$

Lemma 4.93 (TC4-idempotent).

$$\mathbf{TC4} \circ \mathbf{TC4} = \mathbf{TC4}$$

Proof of Lemma 4.93.

$$\begin{aligned}
&\mathbf{TC4}(\mathbf{TC4}(P)) = && \text{(def. of } \mathbf{TC4}\text{)}\\
&= \mathbf{TC4}(P) \land (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= P \land (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))\land\\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(prop. calculus)}\\
&= P \land (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= \mathbf{TC4}(P)
\end{aligned}$$

Lemma 4.94 (TC5-idempotent).

$$\mathbf{TC5} \circ \mathbf{TC5} = \mathbf{TC5}$$

Proof of Lemma 4.94.

$$\begin{aligned}
&\mathbf{TC5}(\mathbf{TC5}(P)) = && \text{(def. of } \mathbf{TC5}\text{)}\\
&= \mathbf{TC5}(P) \land (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) && \text{(def. of } \mathbf{TC5}\text{)}\\
&= P \land (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))\land\\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) && \text{(prop. calculus)}\\
&= P \land (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) && \text{(def. of } \mathbf{TC5}\text{)}\\
&= \mathbf{TC5}(P)
\end{aligned}$$

Lemma 4.95 (commutativity-TC4-R1).

$$\mathbf{TC4} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC4}$$

Proof of Lemma 4.95.

$$\begin{aligned}
&\mathbf{TC4}(\mathbf{R1}(P)) = && \text{(def. of } \mathbf{R1}\text{)}\\
&= \mathbf{TC4}(P \land (tr \le tr')) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= P \land (tr \le tr')\land(\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(prop. calculus)}\\
&= P \land (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))\land(tr \le tr') && \text{(def. of } \mathbf{TC4}\text{)}\\
&= \mathbf{TC4}(P) \land (tr \le tr') && \text{(def. of } \mathbf{R1}\text{)}\\
&= \mathbf{R1}(\mathbf{TC4}(P))
\end{aligned}$$

Lemma 4.96 (commutativity-TC4-R2).

$$\mathbf{TC4} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC4}$$

Proof of Lemma 4.96.

$\mathbf{TC4}(\mathbf{R2}(P(tr,tr'))) =$ (def. of **R2**)
$= \mathbf{TC4}(P(\langle\rangle, tr' - tr))$ (def. of **TC4**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (tr',tr are not used in **TC4**)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))))(\langle\rangle, tr' - tr)$ (def. of **TC4**)
$= \mathbf{TC4}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{TC4}(P))$

Lemma 4.97 (commutativity-TC4-IOCO1).

$$\mathbf{TC4} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC4}$$

Proof of Lemma 4.97.

$\mathbf{TC4}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{TC4}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **TC4**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge$
$\quad (ok \Rightarrow (wait' \vee ok'))$ (def. of **TC4**)
$= \mathbf{TC4}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{TC4}(P))$

Lemma 4.98 (commutativity-TC4-TC1).

$$\mathbf{TC4} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC4}$$

Proof of Lemma 4.98.

$\mathbf{TC4}(\mathbf{TC1}(P)) =$ (def. of **TC1**)
$= \mathbf{TC4}(P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (def. of **TC4**)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge$
$\quad (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC4**)
$= \mathbf{TC4}(P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= \mathbf{TC1}(\mathbf{TC4}(P))$

Lemma 4.99 (commutativity-TC4-TC2).

$$\mathbf{TC4} \circ \mathbf{TC2} = \mathbf{TC2} \circ \mathbf{TC4}$$

Proof of Lemma 4.99.

$$\begin{aligned}
&\mathbf{TC4}(\mathbf{TC2}(P)) = && \text{(def. of } \mathbf{TC2}\text{)}\\
&= \mathbf{TC4}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge \\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(prop. calculus)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge \\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= \mathbf{TC4}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) && \text{(def. of } \mathbf{TC2}\text{)}\\
&= \mathbf{TC2}(\mathbf{TC4}(P))
\end{aligned}$$

Lemma 4.100 (commutativity-TC4-TC3).

$$\mathbf{TC4} \circ \mathbf{TC3} = \mathbf{TC3} \circ \mathbf{TC4}$$

Proof of Lemma 4.100.

$$\begin{aligned}
&\mathbf{TC4}(\mathbf{TC3}(P)) = && \text{(def. of } \mathbf{TC3}\text{)}\\
&= \mathbf{TC4}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge \\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(prop. calculus)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge \\
&\qquad (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= \mathbf{TC4}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) && \text{(def. of } \mathbf{TC3}\text{)}\\
&= \mathbf{TC3}(\mathbf{TC4}(P))
\end{aligned}$$

Lemma 4.101 (commutativity-TC5-R1).

$$\mathbf{TC5} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC5}$$

Proof of Lemma 4.101.

$$\begin{aligned}
&\mathbf{TC5}(\mathbf{R1}(P)) = && \text{(def. of } \mathbf{R1}\text{)}\\
&= \mathbf{TC5}(P \wedge (tr \leq tr')) && \text{(def. of } \mathbf{TC5}\text{)}\\
&= P \wedge (tr \leq tr') \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) && \text{(prop. calculus)}\\
&= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge (tr \leq tr') && \text{(def. of } \mathbf{TC5}\text{)}\\
&= \mathbf{TC5}(P) \wedge (tr \leq tr') && \text{(def. of } \mathbf{R1}\text{)}\\
&= \mathbf{R1}(\mathbf{TC5}(P))
\end{aligned}$$

Lemma 4.102 (commutativity-TC5-R2).

$$\mathbf{TC5} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC5}$$

Proof of Lemma 4.102.

$\mathbf{TC5}(\mathbf{R2}(P(tr,tr'))) =$ (def. of **R2**)
$= \mathbf{TC5}(P(\langle\rangle, tr' - tr))$ (def. of **TC5**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (tr',tr are not used in **TC5**)
$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))))(\langle\rangle, tr' - tr)$ (def. of **TC5**)
$= \mathbf{TC5}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{TC5}(P))$

Lemma 4.103 (commutativity-TC5-IOCO1).

$$\mathbf{TC5} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC5}$$

Proof of Lemma 4.103.

$\mathbf{TC5}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{TC5}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **TC5**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **TC5**)
$= \mathbf{TC5}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{TC5}(P))$

Lemma 4.104 (commutativity-TC5-TC1).

$$\mathbf{TC5} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC5}$$

Proof of Lemma 4.104.

$\mathbf{TC5}(\mathbf{TC1}(P)) =$ (def. of **TC1**)
$= \mathbf{TC5}(P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (def. of **TC5**)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (prop. calculus)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge$
$\quad (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC5**)
$= \mathbf{TC5}(P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= \mathbf{TC1}(\mathbf{TC5}(P))$

Lemma 4.105 (commutativity-TC5-TC2).

$$\mathbf{TC5} \circ \mathbf{TC2} = \mathbf{TC2} \circ \mathbf{TC5}$$

Proof of Lemma 4.105.

$\mathbf{TC5}(\mathbf{TC2}(P)) =$ (def. of **TC2**)

$= \mathbf{TC5}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))))$ (def. of **TC5**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (def. of **TC5**)

$= \mathbf{TC5}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (def. of **TC2**)

$= \mathbf{TC2}(\mathbf{TC5}(P))$

Lemma 4.106 (commutativity-TC5-TC3).

$$\mathbf{TC5} \circ \mathbf{TC3} = \mathbf{TC3} \circ \mathbf{TC5}$$

Proof of Lemma 4.106.

$\mathbf{TC5}(\mathbf{TC3}(P)) =$ (def. of **TC3**)

$= \mathbf{TC5}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))))$ (def. of **TC5**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (def. of **TC5**)

$= \mathbf{TC5}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (def. of **TC3**)

$= \mathbf{TC3}(\mathbf{TC5}(P))$

Lemma 4.107 (commutativity-TC5-TC4).

$$\mathbf{TC5} \circ \mathbf{TC4} = \mathbf{TC4} \circ \mathbf{TC5}$$

Proof of Lemma 4.107.

$\mathbf{TC5}(\mathbf{TC4}(P)) =$ (def. of **TC4**)

$= \mathbf{TC5}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))))$ (def. of **TC5**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (prop. calculus)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (def. of **TC5**)

$= \mathbf{TC5}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (def. of **TC4**)

$= \mathbf{TC4}(\mathbf{TC5}(P))$

Lemma 4.108 (TC6-idempotent).

$$\mathbf{TC6} \circ \mathbf{TC6} = \mathbf{TC6}$$

Proof of Lemma 4.108.

$\mathbf{TC6}(\mathbf{TC6}(P)) =$ (def. of **TC6**)
$= \mathbf{TC6}(P) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (def. of **TC6**)
$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)
$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (def. of **TC6**)
$= \mathbf{TC6}(P)$

Lemma 4.109 (commutativity-TC6-R1).

$$\mathbf{TC6} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{TC6}$$

Proof of Lemma 4.109.

$\mathbf{TC6}(\mathbf{R1}(P)) =$ (def. of **R1**)
$= \mathbf{TC6}(P \wedge (tr \leq tr'))$ (def. of **TC6**)
$= P \wedge (tr \leq tr') \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)
$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge (tr \leq tr')$ (def. of **TC6**)
$= \mathbf{TC6}(P) \wedge (tr \leq tr')$ (def. of **R1**)
$= \mathbf{R1}(\mathbf{TC6}(P))$

Lemma 4.110 (commutativity-TC6-R2).

$$\mathbf{TC6} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{TC6}$$

Proof of Lemma 4.110.

$\mathbf{TC6}(\mathbf{R2}(P(tr, tr'))) =$ (def. of **R2**)
$= \mathbf{TC6}(P(\langle\rangle, tr' - tr))$ (def. of **TC6**)
$= P(\langle\rangle, tr' - tr) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (tr',tr are not used in **TC6**)
$= (P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')))(\langle\rangle, tr' - tr)$ (def. of **TC6**)
$= \mathbf{TC6}(P)(\langle\rangle, tr' - tr)$ (def. of **R2**)
$= \mathbf{R2}(\mathbf{TC6}(P))$

Lemma 4.111 (commutativity-TC6-IOCO1).

$$\mathbf{TC6} \circ \mathbf{IOCO1} = \mathbf{IOCO1} \circ \mathbf{TC6}$$

Proof of Lemma 4.111.

$\mathbf{TC6}(\mathbf{IOCO1}(P)) =$ (def. of **IOCO1**)
$= \mathbf{TC6}(P \wedge (ok \Rightarrow (wait' \vee ok')))$ (def. of **TC6**)
$= P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)
$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **TC6**)
$= \mathbf{TC6}(P) \wedge (ok \Rightarrow (wait' \vee ok'))$ (def. of **IOCO1**)
$= \mathbf{IOCO1}(\mathbf{TC6}(P))$

Lemma 4.112 (commutativity-TC6-TC1).

$$\mathbf{TC6} \circ \mathbf{TC1} = \mathbf{TC1} \circ \mathbf{TC6}$$

Proof of Lemma 4.112.

$\mathbf{TC6}(\mathbf{TC1}(P)) =$ (def. of **TC1**)
$= \mathbf{TC6}(P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (def. of **TC6**)
$= P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)
$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC6**)
$= \mathbf{TC6}(P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)
$= \mathbf{TC1}(\mathbf{TC6}(P))$

Lemma 4.113 (commutativity-TC6-TC2).

$$\mathbf{TC6} \circ \mathbf{TC2} = \mathbf{TC2} \circ \mathbf{TC6}$$

Proof of Lemma 4.113.

$\mathbf{TC6}(\mathbf{TC2}(P)) =$ (def. of **TC2**)
$= \mathbf{TC6}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))))$ (def. of **TC6**)
$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge$
$\quad (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)
$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (def. of **TC6**)
$= \mathbf{TC6}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (def. of **TC2**)
$= \mathbf{TC2}(\mathbf{TC6}(P))$

Lemma 4.114 (commutativity-TC6-TC3).

$$\mathbf{TC6} \circ \mathbf{TC3} = \mathbf{TC3} \circ \mathbf{TC6}$$

Proof of Lemma 4.114.

$\mathbf{TC6}(\mathbf{TC3}(P)) =$ (def. of **TC3**)

$= \mathbf{TC6}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))))$ (def. of **TC6**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)

$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (def. of **TC6**)

$= \mathbf{TC6}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ (def. of **TC3**)

$= \mathbf{TC3}(\mathbf{TC6}(P))$

Lemma 4.115 (commutativity-TC6-TC4).

$$\mathbf{TC6} \circ \mathbf{TC4} = \mathbf{TC4} \circ \mathbf{TC6}$$

Proof of Lemma 4.115.

$\mathbf{TC6}(\mathbf{TC4}(P)) =$ (def. of **TC4**)

$= \mathbf{TC6}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))))$ (def. of **TC6**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge$
$\quad (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)

$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (def. of **TC6**)

$= \mathbf{TC6}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ (def. of **TC4**)

$= \mathbf{TC4}(\mathbf{TC6}(P))$

Lemma 4.116 (commutativity-TC6-TC5).

$$\mathbf{TC6} \circ \mathbf{TC5} = \mathbf{TC5} \circ \mathbf{TC6}$$

Proof of Lemma 4.116.

$\mathbf{TC6}(\mathbf{TC5}(P)) =$ (def. of **TC5**)

$= \mathbf{TC6}(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))))$ (def. of **TC6**)

$= P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge$
$\quad (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (prop. calculus)

$= P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge$
$\quad (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (def. of **TC6**)

$= \mathbf{TC6}(P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (def. of **TC5**)

$= \mathbf{TC5}(\mathbf{TC6}(P))$

Lemma 4.117 (commutativity-TC1-R3^{TC}).

$$\mathbf{TC1} \circ \mathbf{R3}^{TC} = \mathbf{R3}^{TC} \circ \mathbf{TC1}$$

Proof of Lemma 4.117.

$$\begin{array}{ll}
\mathbf{TC1}(\mathbf{R3}^{TC}(P)) = & \text{(def. of } \mathbf{R3}^{TC}\text{)}\\
= \mathbf{TC1}(\mathbb{I}^{TC} \lhd wait \rhd P) & \text{(def. of } \mathbf{TC1}\text{)}\\
= (\mathbb{I}^{TC} \lhd wait \rhd P) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) & (\wedge\text{-if-distr})\\
= (\mathbb{I}^{TC} \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) \lhd wait \rhd & \\
\quad (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) & \text{(def. of } \mathbb{I}^{TC}\text{)}\\
= \left(\begin{array}{l} (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge \\ \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr') \wedge \\ (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge \dots \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \\ (tr' = tr) \wedge \dots \end{array} \right) \right) \end{array} \right) \lhd wait \rhd & \\
\quad (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) & \text{(def. of } \mathbf{TC1} \text{ and prop. calculus)}\\
= \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr') \wedge \\ (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge \dots \wedge \\ (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \\ (tr' = tr) \wedge \dots \wedge \\ (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \end{array} \right) \right) \lhd wait \rhd \mathbf{TC1}(P) & \\
& \text{(prop. calculus and one point rule)}\\
= \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr') \wedge \\ (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge \dots \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \\ (tr' = tr) \wedge \dots \wedge \\ (length(tr' - tr) \leq 0) \end{array} \right) \right) \lhd wait \rhd \mathbf{TC1}(P) & \\
& \text{(def. of length and prop. calculus)}\\
= \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr') \wedge \\ (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge \dots \end{array} \right) \vee \left(\begin{array}{l} ok' \wedge \\ (tr' = tr) \wedge \dots \end{array} \right) \right) \lhd wait \rhd \mathbf{TC1}(P) & \\
& \text{(def. of } \mathbb{I}^{TC}\text{)}\\
= \mathbb{I}^{TC} \lhd wait \rhd (\mathbf{TC1}(P)) & \text{(def. of } \mathbf{R3}^{TC}\text{)}\\
= \mathbf{R3}^{TC}(\mathbf{TC1}(P)) &
\end{array}$$

Lemma 4.118 (commutativity-TC2-R3^{TC}).

$$\mathbf{TC2} \circ \mathbf{R3}^{TC} = \mathbf{R3}^{TC} \circ \mathbf{TC2}$$

Proof of Lemma 4.118.

$$\begin{array}{ll}
\mathbf{TC2}(\mathbf{R3}^{TC}(P)) = & \text{(def. of } \mathbf{R3}^{TC}\text{)}\\
= \mathbf{TC2}(\mathbb{I}^{TC} \lhd wait \rhd P) & \text{(def. of } \mathbf{TC2}\text{)}\\
= (\mathbb{I}^{TC} \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) & (\wedge\text{-if-distr})\\
= (\mathbb{I}^{TC} \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) \lhd wait \rhd & \\
\quad (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) & \text{(def. of if and } \neg wait\text{)}\\
= \mathbb{I}^{TC} \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) & \text{(def. of } \mathbf{TC2}\text{)}\\
= \mathbb{I}^{TC} \lhd wait \rhd (\mathbf{TC2}(P)) & \text{(def. of } \mathbf{R3}^{TC}\text{)}\\
= \mathbf{R3}^{TC}(\mathbf{TC2}(P)) &
\end{array}$$

Lemma 4.119 (commutativity-TC3-R3^{TC}).

$$\mathbf{TC3} \circ \mathbf{R3}^{TC} = \mathbf{R3}^{TC} \circ \mathbf{TC3}$$

Proof of Lemma 4.119.

$\mathbf{TC3}(\mathbf{R3}^{TC}(P)) =$ (def. of $\mathbf{R3}^{TC}$)

$= \mathbf{TC3}(\mathbb{I}^{TC} \lhd wait \rhd P)$ (def. of $\mathbf{TC3}$)

$= (\mathbb{I}^{TC} \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))$ ($\wedge$-if-distr)

$= (\mathbb{I}^{TC} \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) \lhd wait \rhd$
$(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))))$ (def. of if and $\neg wait$)

$= \mathbb{I}^{TC} \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))))$ (def. of $\mathbf{TC3}$)

$= \mathbb{I}^{TC} \lhd wait \rhd (\mathbf{TC3}(P))$ (def. of $\mathbf{R3}^{TC}$)

$= \mathbf{R3}^{TC}(\mathbf{TC3}(P))$

Lemma 4.120 (commutativity-TC4-R3^{TC}).

$$\mathbf{TC4} \circ \mathbf{R3}^{TC} = \mathbf{R3}^{TC} \circ \mathbf{TC4}$$

Proof of Lemma 4.120.

$\mathbf{TC4}(\mathbf{R3}^{TC}(P)) =$ (def. of $\mathbf{R3}^{TC}$)

$= \mathbf{TC4}(\mathbb{I}^{TC} \lhd wait \rhd P)$ (def. of $\mathbf{TC4}$)

$= (\mathbb{I}^{TC} \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))$ ($\wedge$-if-distr)

$= (\mathbb{I}^{TC} \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) \lhd wait \rhd$
$(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))))$ (def. of if and $\neg wait$)

$= \mathbb{I}^{TC} \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))))$ (def. of $\mathbf{TC4}$)

$= \mathbb{I}^{TC} \lhd wait \rhd (\mathbf{TC4}(P))$ (def. of $\mathbf{R3}^{TC}$)

$= \mathbf{R3}^{TC}(\mathbf{TC4}(P))$

Lemma 4.121 (commutativity-TC5-R3^{TC}).

$$\mathbf{TC5} \circ \mathbf{R3}^{TC} = \mathbf{R3}^{TC} \circ \mathbf{TC5}$$

Proof of Lemma 4.121.

$\mathbf{TC5}(\mathbf{R3}^{TC}(P)) =$ (def. of $\mathbf{R3}^{TC}$)

$= \mathbf{TC5}(\mathbb{I}^{TC} \lhd wait \rhd P)$ (def. of $\mathbf{TC5}$)

$= (\mathbb{I}^{TC} \lhd wait \rhd P) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ ($\wedge$-if-distr)

$= (\mathbb{I}^{TC} \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))) \lhd wait \rhd$
$(P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))))$ (def. of if and $\neg wait$)

$= \mathbb{I}^{TC} \lhd wait \rhd (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))))$ (def. of $\mathbf{TC5}$)

$= \mathbb{I}^{TC} \lhd wait \rhd (\mathbf{TC5}(P))$ (def. of $\mathbf{R3}^{TC}$)

$= \mathbf{R3}^{TC}(\mathbf{TC5}(P))$

Lemma 4.122 (commutativity-TC6-R3^{TC}).

$$\mathbf{TC6} \circ \mathbf{R3}^{TC} = \mathbf{R3}^{TC} \circ \mathbf{TC6}$$

Proof of Lemma 4.122.

$$\begin{aligned}
&\mathbf{TC6}(\mathbf{R3}^{TC}(P)) = && \text{(def. of } \mathbf{R3}^{TC}\text{)}\\
&= \mathbf{TC6}(\mathbb{I}^{TC} \lhd wait \rhd P) && \text{(def. of } \mathbf{TC6}\text{)}\\
&= (\mathbb{I}^{TC} \lhd wait \rhd P) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))) && (\wedge\text{-if-distr})\\
&= (\mathbb{I}^{TC} \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')))) \lhd wait \rhd (P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')))) && \text{(def. of if and } \neg wait\text{)}\\
&= \left(\begin{array}{l} (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))\wedge \\ \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr')\wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge \ldots \end{array}\right) \vee \left(\begin{array}{l} ok'\wedge \\ (wait' = wait) \wedge \ldots \end{array}\right)\right) \end{array}\right) \lhd wait \rhd \\
&\quad (P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))) && \text{(def. of } \mathbf{TC6} \text{ and prop. calculus)}\\
&= \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr')\wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge \cdots \wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \end{array}\right) \vee \left(\begin{array}{l} ok'\wedge \\ (wait' = wait) \wedge \cdots \wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \end{array}\right)\right) \lhd wait \rhd \mathbf{TC6}(P)\\
& && \text{(prop. calculus and def. of if)}\\
&= \left(\begin{array}{l} wait\wedge \\ \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr')\wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge \ldots \end{array}\right) \vee \left(\begin{array}{l} ok'\wedge \\ (wait' = wait) \wedge \cdots \wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \end{array}\right)\right) \end{array}\right) \lhd wait \rhd \mathbf{TC6}(P)\\
& && \text{(prop. calculus)}\\
&= \left(\left(\begin{array}{l} wait \wedge \neg ok \wedge (tr \leq tr')\wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge \ldots \end{array}\right) \vee \left(\begin{array}{l} wait \wedge ok'\wedge \\ (wait' = wait) \wedge \cdots \wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \end{array}\right)\right) \lhd wait \rhd \mathbf{TC6}(P)\\
& && \text{(prop. calculus)}\\
&= \left(\left(\begin{array}{l} wait \wedge \neg ok \wedge (tr \leq tr')\wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge \ldots \end{array}\right) \vee \left(\begin{array}{l} wait \wedge ok'\wedge \\ (wait' = wait) \wedge \ldots \end{array}\right)\right) \lhd wait \rhd \mathbf{TC6}(P)\\
& && \text{(prop. calculus and def. of if)}\\
&= \left(\left(\begin{array}{l} \neg ok \wedge (tr \leq tr')\wedge \\ (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge \ldots \end{array}\right) \vee \left(\begin{array}{l} ok'\wedge \\ (wait' = wait) \wedge \ldots \end{array}\right)\right) \lhd wait \rhd \mathbf{TC6}(P)\\
& && \text{(def. of } \mathbb{I}^{TC}\text{)}\\
&= \mathbb{I}^{TC} \lhd wait \rhd (\mathbf{TC6}(P)) && \text{(def. of } \mathbf{R3}^{TC}\text{)}\\
&= \mathbf{R3}^{TC}(\mathbf{TC6}(P))
\end{aligned}$$

Lemma 4.123 (closure-;-TC1).

$$\mathbf{TC1}(P;Q) = P;Q \text{ provided P and Q are } \mathbf{TC1} \text{ healthy}$$

Proof of Lemma 4.123.

$\mathbf{TC1}(P;Q) =$ (def. of ;)

$= \mathbf{TC1}(\exists v_0, tr_0 \bullet P[v_0, tr_0/v', tr'] \wedge Q[v_0, tr_0/v, tr])$ (def. of **TC1**)

$= \exists v_0, tr_0 \bullet P[v_0, tr_0/v', tr'] \wedge Q[v_0, tr_0/v, tr] \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (assumption)

$= \exists v_0, tr_0 \bullet \mathbf{TC1}(P)[v_0, tr_0/v', tr'] \wedge \mathbf{TC1}(Q)[v_0, tr_0/v, tr] \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)

$= \exists v_0, tr_0 \bullet (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))[v_0, tr_0/v', tr'] \wedge$
$\quad (Q \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))[v_0, tr_0/v, tr] \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of [])

$= \exists v_0, tr_0 \bullet P[v_0, tr_0/v', tr'] \wedge (\exists n \in \mathbb{N} \bullet length(tr_0 - tr) \leq n) \wedge$
$\quad Q[v_0, tr_0/v, tr] \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr_0) \leq n) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of $\leq$ and -)

$= \exists v_0, tr_0 \bullet P[v_0, tr_0/v', tr'] \wedge (\exists n \in \mathbb{N} \bullet length(tr_0 - tr) \leq n) \wedge$
$\quad Q[v_0, tr_0/v, tr] \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr_0) \leq n)$ (def. of [])

$= \exists v_0, tr_0 \bullet (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))[v_0, tr_0/v', tr'] \wedge$
$\quad (Q \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))[v_0, tr_0/v, tr] \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (def. of **TC1**)

$= \exists v_0, tr_0 \bullet \mathbf{TC1}(P)[v_0, tr_0/v', tr'] \wedge \mathbf{TC1}(Q)[v_0, tr_0/v, tr]$ (assumption)

$= \exists v_0, tr_0 \bullet P[v_0, tr_0/v', tr'] \wedge Q[v_0, tr_0/v, tr]$ (def. of ;)

$= P;Q$

Lemma 4.124 (closure-;-TC2).

$$\mathbf{TC2}(P;Q) = P;Q \text{ provided that P and Q are } \mathbf{TC2} \text{ healthy and Q is } \mathbf{R3}^{\delta}_{\iota} \text{ healthy}$$

Proof of Lemma 4.124.

$\mathbf{TC2}(P;Q) =$ (assumption and def. of $\mathbf{R3}^{TC}$ and ;)

$= \mathbf{TC2}(\exists v_0 \bullet P[v_0/v'] \land (\mathbb{I}^{TC} \lhd wait \rhd Q)[v_0/v])$ (def. $\mathbf{TC2}$, if, and substitution)

$= \exists v_0 \bullet P[v_0/v'] \land (\mathbb{I}^{TC}[v_0/v] \land wait_0 \lor \neg wait_0 \land Q[v_0/v]) \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset)$

(def. of $\mathbb{I}^{TC}$ and prop. calculus)

$$= \exists v_0 \bullet \left(\left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \land \\ \left(\begin{array}{l} \neg ok \land \cdots \land (\neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \lor \\ ok' \land \cdots \land (wait' = wait) \land (ref'_{in} = ref_{in}) \end{array} \right) [v_0/v] \land wait_0 \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \end{array} \right) \land \left(\begin{array}{l} wait \lor \\ \neg wait' \lor \\ ref'_{in} = \mathcal{A}_{in} \\ \lor ref'_{in} = \emptyset \end{array} \right) \right)$$

(substitution and prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land \\ (\neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right) \lor \\ \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \land \\ (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land (\neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \lor \\ \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \land \\ (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right)$$

(substitution)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land (\neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \lor \\ \left(\begin{array}{l} (P \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset))[v_0/v'] \land wait_0 \land \\ ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \end{array} \right) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right)$$

(def. of $\mathbf{TC2}$ and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \neg ok_0 \land \cdots \land (\neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \lor \\ P[v_0/v'] \land wait_0 \land ok' \land \cdots \land (wait' = wait_0) \land (ref'_{in} = ref_{in0}) \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right)$$

(prop. calculus and def. of $\mathbb{I}^{TC}$ and assumption)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \mathbb{I}^{TC}[v_0/v] \lor \\ \mathbf{TC2}(P)[v_0/v'] \land \neg wait_0 \land \mathbf{TC2}[v_0/v] \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right)$$

(def. $\mathbf{TC2}$ and renaming)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \mathbb{I}^{TC}[v_0/v] \lor \\ \left(\begin{array}{l} P[v_0/v'] \land (wait \lor \neg wait_0 \lor ref_{in0} = \mathcal{A}_{in} \lor ref_{in0} = \emptyset) \land \neg wait_0 \land \\ Q[v_0/v] \land (wait_0 \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \land \\ (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right) \end{array} \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(P[v_0/v'] \land wait_0 \land \mathbb{I}^{TC}[v_0/v] \lor \left(\begin{array}{l} P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (\neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \land \\ (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right) \right)$$

(prop. calculus)

$$= \exists v_0 \bullet \left(\begin{array}{l} P[v_0/v'] \land wait_0 \land \mathbb{I}^{TC}[v_0/v] \lor \\ P[v_0/v'] \land \neg wait_0 \land Q[v_0/v] \land (wait \lor \neg wait' \lor ref'_{in} = \mathcal{A}_{in} \lor ref'_{in} = \emptyset) \end{array} \right)$$

(substitution and def. of $\mathbf{TC2}$)

$= \exists v_0 \bullet P[v_0/v'] \land wait_0 \land \mathbb{I}^{TC}[v_0/v] \lor P[v_0/v'] \land \neg wait_0 \land \mathbf{TC2}(Q)[v_0/v]$ (assumption and def. of $\mathbf{R3}^{TC}$)

$= \exists v_0 \bullet P[v_0/v'] \land \mathbf{R3}^{TC}(Q)[v_0/v]$ (assumption and def. of ;)

$= P;Q$

Lemma 4.125 (closure-;-TC3).

$$\mathbf{TC3}(P;Q) = P;Q \text{ provided Q is } \mathbf{TC3} \text{ healthy}$$

Proof of Lemma 4.125. Similar to the proof of Lemma 4.124.

Lemma 4.126 (closure-;-TC4).

$$\mathbf{TC4}(P;Q) = P;Q \text{ provided Q is } \mathbf{TC4} \text{ healthy}$$

Proof of Lemma 4.126. Similar to the proof of Lemma 4.124.

Lemma 4.127 (closure-;-TC5).

$$\mathbf{TC5}(P;Q) = P;Q \text{ provided Q is } \mathbf{TC5} \text{ healthy}$$

Proof of Lemma 4.127. Similar to the proof of Lemma 4.124.

Lemma 4.128 (closure-;-TC6).

$$\mathbf{TC6}(P;Q) = P;Q \text{ provided Q is } \mathbf{TC6} \text{ healthy}$$

Proof of Lemma 4.128. Similar to the proof of Lemma 4.124.

Lemma 4.129 (closure-+-IOCO1).

$$\mathbf{IOCO1}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{IOCO1} \text{ healthy}$$

Proof of Lemma 4.129.

$$\begin{aligned}
&\mathbf{IOCO1}(P+Q) = && \text{(def. of } +\text{)}\\
&= \mathbf{IOCO1}((P \wedge Q) \lhd \delta \rhd (P \vee Q)) && \text{(def. of } \mathbf{IOCO1}\text{)}\\
&= (P \wedge Q) \lhd \delta \rhd (P \vee Q)) \wedge (ok \Rightarrow (wait' \vee ok')) && \text{(distr. of } \wedge \text{ over if)}\\
&= (P \wedge Q \wedge (ok \Rightarrow (wait' \vee ok'))) \lhd \delta \rhd \\
&\qquad ((P \vee Q) \wedge (ok \Rightarrow (wait' \vee ok'))) && \text{(prop. calculus)}\\
&= (P \wedge (ok \Rightarrow (wait' \vee ok')) \wedge Q \wedge (ok \Rightarrow (wait' \vee ok'))) \lhd \delta \rhd \\
&\qquad ((P \wedge (ok \Rightarrow (wait' \vee ok'))) \vee (Q \wedge (ok \Rightarrow (wait' \vee ok')))) && \text{(def. of } +\text{)}\\
&= (P \wedge (ok \Rightarrow (wait' \vee ok'))) + (Q \wedge (ok \Rightarrow (wait' \vee ok'))) && \text{(def. of } \mathbf{IOCO1}\text{)}\\
&= \mathbf{IOCO1}(P) + \mathbf{IOCO1}(Q) && \text{(assumption)}\\
&= P+Q
\end{aligned}$$

Lemma 4.130 (closure-+-TC1).

$$\mathbf{TC1}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{TC1} \text{ healthy}$$

Proof of Lemma 4.130.

$\mathbf{TC1}(P+Q) =$ (def. of $+$)

$= \mathbf{TC1}((P \wedge Q) \lhd \delta \rhd (P \vee Q))$ (def. of **TC1**)

$= (P \wedge Q) \lhd \delta \rhd (P \vee Q)) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)$ (distr. of $\wedge$ over if)

$= (P \wedge Q \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) \lhd \delta \rhd$
$\quad ((P \vee Q) \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (prop. calculus)

$= (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n) \wedge$
$\quad Q \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) \lhd \delta \rhd$
$\quad ((P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) \vee (Q \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)))$ (def. of $+$)

$= (P \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n)) + (Q \wedge (\exists n \in \mathbb{N} \bullet length(tr' - tr) \leq n))$ (def. of **TC1**)

$= \mathbf{TC1}(P) + \mathbf{TC1}(Q)$ (assumption)

$= P + Q$

Lemma 4.131 (closure-+-TC2).

$$\mathbf{TC2}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{TC2} \text{ healthy}$$

Proof of Lemma 4.131.

$\mathbf{TC2}(P+Q) =$ (def. of $+$)

$= \mathbf{TC2}((P \wedge Q) \lhd \delta \rhd (P \vee Q))$ (def. of **TC2**)

$= (P \wedge Q) \lhd \delta \rhd (P \vee Q)) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))$ (distr. of $\wedge$ over if)

$= (P \wedge Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) \lhd \delta \rhd$
$\quad ((P \vee Q) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))))$ (prop. calculus)

$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))) \wedge$
$\quad Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) \lhd \delta \rhd$
$\quad ((P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) \vee$
$\quad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))))$ (def. of $+$)

$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset)))) +$
$\quad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (ref'_{in} = \mathcal{A}_{in} \vee ref'_{in} = \emptyset))))$ (def. of **TC2**)

$= \mathbf{TC2}(P) + \mathbf{TC2}(Q)$ (assumption)

$= P + Q$

Lemma 4.132 (closure-$+$-TC3).

$$\mathbf{TC3}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{TC3} \text{ healthy}$$

Proof of Lemma 4.132.

$$\begin{aligned}
&\mathbf{TC3}(P+Q) = && \text{(def. of } +\text{)}\\
&= \mathbf{TC3}((P \wedge Q) \lhd \delta \rhd (P \vee Q)) && \text{(def. of } \mathbf{TC3}\text{)}\\
&= (P \wedge Q) \lhd \delta \rhd (P \vee Q)) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) && \text{(distr. of } \wedge \text{ over if)}\\
&= (P \wedge Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) \lhd \delta \rhd\\
&\qquad ((P \vee Q) \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) && \text{(prop. calculus)}\\
&= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))) \wedge\\
&\qquad Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) \lhd \delta \rhd\\
&\qquad ((P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) \vee\\
&\qquad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1))))) && \text{(def. of } +\text{)}\\
&= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) +\\
&\qquad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow (|ref'_{out}| \geq |\mathcal{A}_{out}| - 1)))) && \text{(def. of } \mathbf{TC3}\text{)}\\
&= \mathbf{TC3}(P) + \mathbf{TC3}(Q) && \text{(assumption)}\\
&= P + Q
\end{aligned}$$

Lemma 4.133 (closure-$+$-TC4).

$$\mathbf{TC4}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{TC4} \text{ healthy}$$

Proof of Lemma 4.133.

$$\begin{aligned}
&\mathbf{TC4}(P+Q) = && \text{(def. of } +\text{)}\\
&= \mathbf{TC4}((P \wedge Q) \lhd \delta \rhd (P \vee Q)) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= (P \wedge Q) \lhd \delta \rhd (P \vee Q))\\
&\qquad \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) && \text{(distr. of } \wedge \text{ over if)}\\
&= (P \wedge Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) \lhd \delta \rhd\\
&\qquad ((P \vee Q) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) && \text{(prop. calculus)}\\
&= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))) \wedge\\
&\qquad Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) \lhd \delta \rhd\\
&\qquad ((P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) \vee\\
&\qquad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in}))))) && \text{(def. of } +\text{)}\\
&= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) +\\
&\qquad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((|ref'_{out}| = |\mathcal{A}_{out}| - 1) \iff ref'_{in} = \mathcal{A}_{in})))) && \text{(def. of } \mathbf{TC4}\text{)}\\
&= \mathbf{TC4}(P) + \mathbf{TC4}(Q) && \text{(assumption)}\\
&= P + Q
\end{aligned}$$

Lemma 4.134 (closure-$+$-TC5).

$$\mathbf{TC5}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{TC5} \text{ healthy}$$

Proof of Lemma 4.134.

$\mathbf{TC5}(P+Q) =$ (def. of $+$)

$= \mathbf{TC5}((P \wedge Q) \lhd \delta \rhd (P \vee Q))$ (def. of $\mathbf{TC5}$)

$= (P \wedge Q) \lhd \delta \rhd (P \vee Q))$
$\quad \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))$ (distr. of $\wedge$ over if)

$= (P \wedge Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))) \lhd \delta \rhd$
$\quad ((P \vee Q) \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))))$ (prop. calculus)

$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))) \wedge$
$\quad Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))) \lhd \delta \rhd$
$\quad ((P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))) \vee$
$\quad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))))$ (def. of $+$)

$= (P \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out})))) +$
$\quad (Q \wedge (\neg wait \Rightarrow (wait' \Rightarrow ((ref'_{in} = \emptyset) \iff ref'_{out} = \mathcal{A}_{out}))))$ (def. of $\mathbf{TC5}$)

$= \mathbf{TC5}(P) + \mathbf{TC5}(Q)$ (assumption)

$= P+Q$

Lemma 4.135 (closure-$+$-TC6).

$$\mathbf{TC6}(P+Q) = P+Q \text{ provided P and Q are } \mathbf{TC6} \text{ healthy}$$

Proof of Lemma 4.135.

$\mathbf{TC6}(P+Q) =$ (def. of $+$)

$= \mathbf{TC6}((P \wedge Q) \lhd \delta \rhd (P \vee Q))$ (def. of $\mathbf{TC6}$)

$= (P \wedge Q) \lhd \delta \rhd (P \vee Q)) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))$ (distr. of $\wedge$ over if)

$= (P \wedge Q \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))) \lhd \delta \rhd$
$\quad ((P \vee Q) \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')))$ (prop. calculus)

$= (P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')) \wedge Q \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))) \lhd \delta \rhd$
$\quad ((P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))) \vee (Q \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))))$ (def. of $+$)

$= (P \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail'))) + (Q \wedge (\neg wait' \Rightarrow (pass' \Rightarrow \neg fail')))$ (def. of $\mathbf{TC6}$)

$= \mathbf{TC6}(P) + \mathbf{TC6}(Q)$ (assumption)

$= P+Q$

Lemma 4.136 ($\rceil|_U$ -$\sqcap$-distributivity).

$$(T_1 \sqcap T_2)\rceil|_U\ P = (T_1\rceil|_U\ P) \sqcap (T_2\rceil|_U\ P)$$

Proof of Lemma 4.136.

$$\begin{aligned}
&(T_1 \sqcap T_2)\rceil|_U\ P = && \text{(definition of } \rceil|_U \text{)}\\
&= ((\overline{T_1 \sqcap T_2}) \text{⅄} P);M_{ti} && \text{(definition of renaming operator)}\\
&= ((\overline{T_1} \sqcap \overline{T_2}) \text{⅄} P);M_{ti} && \text{(commutativity of } \sqcap \text{ over ⅄)}\\
&= ((\overline{T_1} \text{⅄} P) \sqcap (\overline{T_2} \text{⅄} P));M_{ti} && \text{(commutativity of ; over } \sqcap\text{)}\\
&= (\overline{T_1} \text{⅄} P);M_{ti} \sqcap (\overline{T_2} \text{⅄} P);M_{ti} && \text{(definition of } \rceil|_U \text{)}\\
&= (T_1\rceil|_U\ P) \sqcap (T_2\rceil|_U\ P)
\end{aligned}$$

A.2 Chapter 6 (Test Purpose Based Test Case Selection)

A.2.1 Section 6.3 (Multiple Test Cases per Test Purpose)

Lemma 6.1 (Finite Superposition).
The number of states of the superposition of observers and a complete test graph is finite.
Proof of Lemma 6.1.
In order to show that the number of states if finite, we give an upper bound for the number of states. Let $CTG = (Q, L, \rightarrow, q_0)$ be a complete test graph. The set of states Q is partitioned into verdict states $V = \mathit{Pass} \cup \mathit{Fail} \cup \mathit{Inconclusive}$ and non-verdict states Q', i.e. $Q' \cap V = \emptyset$ and $Q = Q' \cup V$.
A single state of the superposition consists of k triples $(\langle T_1 \rangle, \dots \langle T_k \rangle)$. Each triple T_i comprises a state $q \in Q$ of the complete test graph and the list C holding the current state for every observer.
Given the structure of our observers, a single state $q \in Q$ of the CTG can only be combined with two different states of the observer. Either q is a verdict state ($q \in V$) and can be combined with C or D of the observer or q is not a verdict state and thus can be combined with A or B.
Thus, for n observers we have $t = 2^n * |Q|$ different triples. A single state of the superposition may comprise an arbitrary number of such triples but no duplicates. Hence, the total number of states is given by counting all possible triple combinations.
Given t triples, then the number of triples that can be build by using i different triples is given by $\binom{t}{i}$. The number of all possible combinations can be calculated by

$$\sum_{i=1}^{t} \binom{t}{i} = \sum_{i=1}^{t} \frac{t!}{i! * (t-i)!} = \frac{(2^t - 1) * (t)!}{\gamma(1+t)} < \frac{2^t * t^t}{\frac{2}{3}} < \frac{3}{2} t^{2*t}.$$

This gives an upper bound of $O((2^n * |Q|)^{(2^n * |Q|)})$ for the number of states for the superposition.

Lemma 6.2 (Reachable verdict).
A verdict state is reachable from each state of a generated test case.
Proof of Lemma 6.2.
A test case is generated if and only if all its branches lead to accepting states of the corresponding observer. Assume, that a test case comprises a state which does not lead to a verdict. This implies that there exists a path within the observer leading to an accepting state, but where no guard checks for a verdict state. This contradicts the structure of observers (Figure 6.4) since every path to the accepting state is guarded by a check for verdict states, i.e., $s' \in V$.

Lemma 6.3 (Input enabled).
The generated test cases are input complete in all states where inputs are possible.
Proof of Lemma 6.3.
If there are input edges within the complete test graph, then the lines 20 to 24 ensure that all input edges are copied to the generated test case. Assume that a generated test case is not input complete in a state where inputs are enabled. This can only be the case if the complete test graph (CTG) is not input complete in its corresponding state. However, this contradicts the definition of a CTG (Definition 6.2), which requires input-completeness in all states where inputs are possible.

Lemma 6.4 (Controllability).
Every generated test case is controllable.
Proof of Lemma 6.4.
Assume, that a generated test case is not controllable, i.e., the test case contains a state q where both inputs and outputs are enabled. Since every edge in the test case leads to a new (unused) state (lines 7 and 21), this implies that ω' used for outputs (Line 10) is the same test case as the ω' used for inputs (Line 23). However, ω' is re-initialized in Line 19. Thus, the two ω' are different. This contradicts with our assumption.

Bibliography

ABRAN, A. AND MOORE, J. W., Eds. 2004. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Washington, DC, USA.

ABREU, R., ZOETEWEIJ, P., AND VAN GEMUND, A. J. 2007. On the accuracy of spectrum-based fault localization. In *Proceedings of Testing: Academia and Industry Conference - Practice And Research Techniques*, P. McMinn, Ed. IEEE Computer Society, Washington, DC, USA, 89–98.

AICHERNIG, B. K. AND CORRALES DELGADO, C. 2006. From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 3922. Springer, Berlin / Heidelberg, Germany, 324–338.

AICHERNIG, B. K. AND HE, J. 2009. Mutation testing in UTP. *Formal Aspects of Computing 21,* 1-2 (February), 33–64.

AICHERNIG, B. K., PEISCHL, B., WEIGLHOFER, M., AND WOTAWA, F. 2007. Protocol conformance testing a SIP registrar: An industrial application of formal methods. In *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, M. Hinchey and T. Margaria, Eds. IEEE Computer Society, Washington, DC, USA, 215–224.

AICHERNIG, B. K., WEIGLHOFER, M., AND WOTAWA, F. 2008. Improving fault-based conformance testing. *Electronic Notes in Theoretical Computer Science 220,* 1 (December), 63–77.

ALBERTS, D. S. 1976. The economics of software quality assurance. In *Proceedings of the National ComputerConference and Exposition*. ACM, New York, NY, USA, 433–442.

ALUR, R., COURCOUBETIS, C., AND DILL, D. L. 1990. Model-checking for real-time systems. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 414–425.

ALUR, R. AND DILL, D. L. 1994. A theory of timed automata. *Theoretical Computer Science 126,* 2, 183–235.

AMMANN, P., BLACK, P., AND MAJURSKI, W. 1998. Using model checking to generate tests from specifications. In *Proceedings of the 2nd InternationalConference on Formal Engineering Methods*. IEEE Computer Society, Washington, DC, USA, 46–54.

AMMANN, P. AND OFFUTT, J. 2008. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA.

AMMANN, P., OFFUTT, J., AND HUANG, H. 2003. Coverage criteria for logical expressions. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 99–107.

AMYOT, D. AND LOGRIPPO, L. 2000. Structural coverage for lotos - a probe insertion technique. In *Proceedings of the 13th International Conference on Testing Communicating Systems: Tools and Techniques*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 19–34.

AMYOT, D., LOGRIPPO, L., AND WEISS, M. 2005. Generation of test purposes from use case maps. *Computer Networks 49,* 5 (December), 643–660.

AVIŽIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 11–33.

BAETEN, J. C. M. AND WEIJLAND, W. P. 1990. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, UK.

BEIZER, B. 1990. *Software Testing Techniques*. Van Nost. Reinhold, U.S.

BELINFANTE, A., FEENSTRA, J., DE VRIES, R. G., TRETMANS, J., GOGA, N., FEIJS, L. M. G., MAUW, S., AND HEERINK, L. 1999. Formal test automation: A simple experiment. In *Proceedings of the 12th International Workshop on Testing Communicating Systems*. IFIP Conference Proceedings, vol. 147. Kluwer Academic Publishers, Dordrecht, The Netherlands, 179–196.

BERGSTRA, J. A. AND KLOP, J. W. 1985. Algebra of communicating processes with abstraction. *Theoretical Computer Science 37*, 77–121.

BERNOT, G. 1991. Testing against formal specifications: A theoretical view. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, S. Abramsky and T. S. E. Maibaum, Eds. Lecture Notes in Computer Science, vol. 494. Springer, Berlin / Heidelberg, Germany, 99–119.

BIDOIT, M. AND MOSSES, P. 2003. *CASL User Manual: Introduction to Using the Common Algebraic Specication Language*. Lecture Notes in Computer Science, vol. 2900. Springer, Berlin / Heidelberg, Germany.

BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., AND ZHU, Y. 2003. Bounded model checking. *Advances in Computers 58*, 118–149.

BLACK, P. E., OKUN, V., AND YESHA, Y. 2000. Mutation operators for specifications. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA, 81–88.

BLOM, J., HESSEL, A., JONSSON, B., AND PETTERSSON, P. 2004. Specifying and generating test cases using observer automata. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing*, J. Grabowski and B. Nielsen, Eds. Lecture Notes in Computer Science, vol. 3395. Springer, Berlin / Heidelberg, Germany, 125–139.

BOHNENKAMP, H. C. AND BELINFANTE, A. 2005. Timed testing with torx. In *Proceedings of the International Symposium of Formal Methods Europe*, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds. Lecture Notes in Computer Science, vol. 3582. Springer, Berlin / Heidelberg, Germany, 173–188.

BOLOGNESI, T. AND BRINKSMA, E. 1987. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems 14,* 1, 25–59.

BORODAY, S., PETRENKO, A., AND GROZ, R. 2007. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science 190,* 2, 3–19.

BOTINČAN, M. AND NOVAKOVIĆ, V. 2007. Model-based testing of the conference protocol with spec explorer. In *Proceedings of the 9th International Conference on Telecommunications*. IEEE, Washington, DC, USA, 131–138.

BOURDONOV, I. B., KOSSATCHEVA, A. S., AND KULIAMIN, V. V. 2006. Formal conformance testing of systems with refused inputs and forbidden actions. *Electronic Notes in Theoretical Computer Science 164,* 4 (October), 83–96.

BRINKSMA, H., HEERINK, A. W., AND TRETMANS, J. 1998. Factorized test generation for multi input/output transition systems. In *Proceedings of the IFIP 11th International Workshop on Testing Communicating Systems*, A. Petrenko and N. Yevtushenko, Eds. IFIP Conference Proceedings, vol. 131. Kluwer Academic Publishers, Dordrecht, The Netherlands, 67–82.

BRIONES, L. B. AND BRINKSMA, E. 2004. A test generation framework for quiescent real-time systems. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing*, J. Grabowski and B. Nielsen, Eds. Lecture Notes in Computer Science, vol. 3395. Springer, Berlin / Heidelberg, Germany, 64–78.

BRIONES, L. B. AND BRINKSMA, E. 2005. Testing real-time multi input-output systems. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, K.-K. Lau and R. Banach, Eds. Lecture Notes in Computer Science, vol. 3785. Springer, Berlin / Heidelberg, Germany, 264–279.

BRIONES, L. B., BRINKSMA, E., AND STOELINGA, M. 2006. A semantic framework for test coverage. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis*. Lecture Notes in Computer Science, vol. 4218. Springer, Berlin / Heidelberg, Germany, 399–414.

BROY, M., JONSSON, B., KATOEN, J.-P., LEUCKER, M., AND PRETSCHNER, A., Eds. 2005. *Model-Based Testing of Reactive Systems*. Lecture Notes in Computer Science, vol. 3472. Springer, Berlin / Heidelberg, Germany.

BRUDA, S. D. 2005. *Preorder Relations*, Chapter 5, 117–149, of Broy et al. (2005).

BUDD, T. A., DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1980. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 220–233.

BUDD, T. A. AND GOPAL, A. S. 1985. Program testing by specification mutation. *Computer languages 10,* 1, 63–73.

BUHR, R. AND CASSELMAN, R. 1996. *Use Case Maps for Objectoriented Systems.* Prentice Hall, Upper Saddle River, NJ, USA.

BURGUILLO-RIAL, J. C., FERNÁNDEZ-IGLESIAS, M. J., GONZÁLEZ-CASTAÑO, F. J., AND LLAMAS-NISTAL, M. 2002. Heuristic-driven test case selection from formal specifications. a case study. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*. Lecture Notes in Computer Science, vol. 2391/2002. Springer, Berlin / Heidelberg, Germany, 141–148.

BUXTON, J. AND RANDELL, B., Eds. 1970. *Software Engineering Techniques – Report on a conference sponsored by the NATO Science Committee*. Rome, Italy. p. 21.

CACCIARI, L. AND RAFIQ, O. 1999. Controllability and observability in distributed testing. *Information and Software Technology 41,* 11 (September), 767–780.

CAMPBELL, C., GRIESKAMP, W., NACHMANSON, L., SCHULTE, W., TILLMANN, N., AND VEANES, M. 2005. Model-based testing of object-oriented reactive systems with spec explorer. Tech. Rep. MSR-TR-2005-59, Microsoft Research. May.

CAVALCANTI, A., HARWOOD, W., AND WOODCOCK, J. 2006. Pointers and records in the unifying theories of programming. In *Proceedings of the 1st International Symposium onUnifying Theories of Programming*, S. Dunne and B. Stoddart, Eds. Lecture Notes in Computer Science, vol. 4010. Springer, Berlin / Heidelberg, Germany, 200–216.

CAVALCANTI, A. AND WOODCOCK, J. 2004. A tutorial introduction to csp in unifying theories of programming. In *Proceedings of the 1st Pernambuco SummerSchool on Software Engineering - Refinement Techniques in Software Engineering*, A. Cavalcanti, A. Sampaio, and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 3167. Springer, Berlin / Heidelberg, Germany, 220–268.

CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. 2002. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 595–604.

CHEUNG, T. AND REN, S. 1993. Executable test sequences with operational coverage for lotos specifications. In *Proceedings of the 12th Annual International Phoenix Conference on Computers and Communications*. IEEE, Washington, DC, USA, 245–253.

CHILENSKI, J. AND MILLER, S. 1994. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal 9,* 5 (September), 193–200.

CLARKE, D., JÉRON, T., RUSU, V., AND ZINOVIEVA, E. 2002. STG: A symbolic test generation tool. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2280. Springer, Berlin / Heidelberg, Germany, 470–475.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8,* 2, 244–263.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 238–252.

CUIJPERS, P. J. L., RENIERS, M. A., AND HEEMELS, W. P. M. H. 2002. Hybrid transition systems. Tech. rep., Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

DA SILVA, D. A. AND MACHADO, P. D. L. 2006. Towards test purpose generation from CTL properties for reactive systems. *Electronic Notes in Theoretical Computer Science 164,* 4, 29–40.

DALLMEIER, V., LINDIG, C., AND ZELLER, A. 2005. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 3586/2005. Springer, Berlin / Heidelberg, Germany, 528–550.

DAVID, A., LARSEN, K. G., LI, S., AND NIELSEN, B. 2008a. Cooperative testing of uncontrollable timed systems. *Electronic Notes in Theoretical Computer Science 220,* 1, 79–92.

DAVID, A., LARSEN, K. G., LI, S., AND NIELSEN, B. 2008b. A game-theoretical approach to real-time system testing. In *Proceedings of the 11th International Conference on Design Automation and Test in Europe*. ACM, New York, NY, USA, 486–491.

DE VRIES, R. G. AND TRETMANS, J. 2001. Towards formal test purposes. In *Proceedings of the International Workshop on Formal Approaches to Testing of Software 2001*, J. Tretmans and H. Brinksma, Eds. BRICS Notes Series, vol. NS-01-4. BRICS Department of Computer Science, Aarhus, Denkmark, 61–76.

DEMILLO, R., LIPTON, R., AND SAYWARD, F. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer 11,* 4 (April), 34–41.

DIJKSTRA, E. W. AND SCHOLTEN, C. S. 1990. *Predicate calculus and program semantics*. Springer-Verlag, New York, NY, USA.

DU BOUSQUET, L., RAMANGALAHY, S., SIMON, S., VIHO, C., BELINFANTE, A., AND DE VRIES, R. G. 2000. Formal test automation: The conference protocol with TGV/TORX. In *Proceedings of 13th International Conference on Testing Communicating Systems: Tools and Techniques*. IFIP Conference Proceedings, vol. 176. Kluwer Academic Publishers, Dordrecht, The Netherlands, 221–228.

DUTERTRE, B. AND DE MOURA, L. 2008. The yices smt solver. Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 - USA. available online: http://yices.csl.sri.com/tool-paper.pdf.

EHRIG, H., FEY, W., AND HANSEN, H. 1983. Act one - an algebraic specification language with two levels of semantics. In *Proceedings 2nd Workshop on Abstract Data Type*, M. Broy and M. Wirsing, Eds.

FAIVRE, A., GASTON, C., GALL, P. L., AND TOUIL, A. 2008. Test purpose concretization through symbolic action refinement. In *Proceedings of the 20th International Conference on Testing of Software and Communicating Systems*, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds. Lecture Notes in Computer Science, vol. 5047. Springer, Berlin / Heidelberg, Germany, 184–199.

FEIJS, L. M. G., GOGA, N., AND MAUW, S. 2000. Probabilities in the torx test derivation algorithm. In *Proceedings of the 2nd Workshop on SDL and MSC*, E. Sherratt, Ed. VERIMAG, IRISA, SDL Forum, Grenoble, France, 173–188.

FEIJS, L. M. G., GOGA, N., MAUW, S., AND TRETMANS, J. 2002. Test selection, trace distance and heuristics. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems*, I. Schieferdecker, H. König, and A. Wolisz, Eds. IFIP Conference Proceedings, vol. 210. Kluwer Academic Publishers, Dordrecht, The Netherlands, 267–282.

FERNANDEZ, J.-C., JARD, C., JÉRON, T., AND VIHO, C. 1997. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming 29,* 1-2, 123–146.

FERNANDEZ, J.-C. AND MOUNIER, L. 1991. "On the fly" verification of behavioural equivalences and preorders. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 575. Springer, Berlin / Heidelberg, Germany, 181–191.

FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., AND STEWART, L. 1999. HTTP authentication: Basic and digest access authentication. RCF 2617, IETF.

FRANTZEN, L., TRETMANS, J., AND WILLEMSE, T. A. C. 2004. Test generation based on symbolic specifications. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing*. Lecture Notes in Computer Science, vol. 3395. Springer, Berlin / Heidelberg, Germany, 1–15.

FRANTZEN, L., TRETMANS, J., AND WILLEMSE, T. A. C. 2006. A symbolic framework for model-based testing. In *Proceedings of the 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*. Lecture Notes in Computer Science, vol. 4262. Springer, Berlin / Heidelberg, Germany, 40–54.

FRASER, G., WEIGLHOFER, M., AND WOTAWA, F. 2008a. Coverage based testing with test purposes. In *Proceedings of the 8th International Conference on Quality Software*. IEEE Computer Society, Washington, DC, USA, 199–208.

FRASER, G., WEIGLHOFER, M., AND WOTAWA, F. 2008b. Using observer automata to select test cases for test purposes. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*. Knowledge Systems Institue Graduate School, Skokie, IL, USA, 709–714.

FRASER, G. AND WOTAWA, F. 2007. Nondeterministic testing with linear model-checker counterexamples. In *Proceedings of the 7th International Conference on Quality Software*. IEEE Computer Society, Washington, DC, USA, 107–116.

FRASER, G., WOTAWA, F., AND AMMANN, P. E. 2009. Testing with model checkers: a survey. *Software Testing, Verification and Reliability 19,* 3 (December), 215–261.

GARAVEL, H. 1998. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Ed. Lecture Notes in Computer Science. Springer, Berlin / Heidelberg, Germany, 68–84.

GARAVEL, H., LANG, F., AND MATEESCU, R. 2002. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter 4*, 13–24.

GARFINKEL, S. 2005. History's worst software bugs. Wired News. available online: `http://www.wired.com/software/coolapps/news/2005/11/69355` (last visited: 15.09.2009).

GARGANTINI, A. AND RICCOBENE, E. 2001. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science 7,* 11, 1050–1067.

GASTON, C., LE GALL, P., RAPIN, N., AND TOUIL, A. 2006. Symbolic execution techniques for test purpose definition. In *Proceedings of the 18th International Conference on Testing of Communicating Systems*. Lecture Notes in Computer Science, vol. 3964. Springer, Berlin / Heidelberg, Germany, 1–18.

GAUDEL, M.-C. 1995. Testing can be formal, too. In *Proceedings of th 6th International Conference on Theory and Practice of Software Development*, P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds. Lecture Notes in Computer Science, vol. 915. Springer, Berlin / Heidelberg, Germany, 82–96.

GOGA, N. 2001. Comparing torx, autolink, tgv and uio test algorithms. In *Proceedings of the 10th International SDL Forum*. Lecture Notes in Computer Science. Springer, Berlin / Heidelberg, Germany, 379–402.

GOGA, N. 2003. Experimenting with the probabilistic torx. In *Proceedings of Software Engineering for High Assurance Systems*. Software Engineering Institute, Portland, OR, USA, 13–20.

GRABOWSKI, J., HOGREFE, D., AND NAHM, R. 1993. Test case generation with test purpose specification by MSC's. In *Proceedings of the 6th SDL Forum*. Elsevier Science Publishers Ltd., Oxford, UK, 253–266.

GRIESKAMP, W., TILLMANN, N., CAMPBELL, C., SCHULTE, W., AND VEANES, M. 2005. Action machines — towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *Proceedings of the International Conference on Software Quality*. IEEE Computer Society, Washington, DC, USA, 72–79.

HAAR, S., JARD, C., AND JOURDAN, G.-V. 2007. Testing input/output partial order automata. In *Proceedings of the 19th International Conference on Testing of Software and Communicating Systems*. Lecture Notes in Computer Science, vol. 4581. Springer, Berlin / Heidelberg, Germany, 171–185.

HAMLET, R. 1977. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering SE-3,* 4 (July), 279–290.

HARROLD, M. J., ROTHERMEL, G., WU, R., AND YI, L. 1998. An empirical investigation of program spectra. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York, NY, USA, 83–90.

HE, J. AND SANDERS, J. W. 2006. Unifying probability. In *Proceedings of the 1st International Symposium on Unifying Theories of Programming*, S. Dunne and B. Stoddart, Eds. Lecture Notes in Computer Science, vol. 4010. Springer, Berlin / Heidelberg, Germany, 173–199.

HEERINK, A. W. AND TRETMANS, J. 1997. Refusal testing for classes of transition systems with inputs and outputs. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, Eds. IFIP Conference Proceedings, vol. 107. Chapman & Hall, London, 23–38.

HEERINK, L., FEENSTRA, J., AND TRETMANS, J. 2000. Formal test automation: The conference protocol with phact. In *Proceedings of the 13th International Conference on Testing Communicating Systems*, H. Ural, R. L. Probert, and G. von Bochmann, Eds. IFIP Conference Proceedings, vol. 176. Kluwer Academic Publishers, Dordrecht, The Netherlands, 211–220.

HENZINGER, T., HO, P.-H., AND WONG-TOI, H. 1997. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer 1*, 110–122.

HESSEL, A., LARSEN, K. G., MIKUCIONIS, M., NIELSEN, B., PETTERSSON, P., AND SKOU, A. 2008. *Testing Real-Time Systems Using UPPAAL*, 77–117, of Hierons et al. (2008).

HESSEL, A. AND PETTERSSON, P. 2007. A global algorithm for model-based test suite generation. *Electronic Notes in Theoretical Computer Science 190,* 2 (August), 47–59.

HIERONS, R. M., BOGDANOV, K., BOWEN, J. P., CLEAVELAND, R., DERRICK, J., DICK, J., GHEORGHE, M., HARMAN, M., KAPOOR, K., KRAUSE, P., LUETTGEN, G., SIMONS, A. J. H., VILKOMIR, S., WOODWARD, M. R., AND ZEDAN, H. 2009. Using formal methods to support testing. *ACM Computing Surveys 41,* 2.

HIERONS, R. M., BOWEN, J. P., AND HARMAN, M., Eds. 2008. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 4949. Springer, Berlin / Heidelberg, Germany.

HIERONS, R. M., MERAYO, M. G., AND NÚÑEZ, M. 2008. Implementation relations for the distributed test architecture. In *Proceedings of the 20th International Conference on Testing of Software and Communicating Systems*, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds. Lecture Notes in Computer Science, vol. 5047. Springer, Berlin / Heidelberg, Germany, 200–215.

HOARE, C. A. R. 1978. Communicating sequential processes. *Communications of the ACM 21,* 8, 666–677.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, USA.

HOARE, C. A. R. AND HE, J. 1998. *Unifying Theories of Programming*. Prentice Hall, Upper Saddle River, NJ, USA.

HOGREFE, D., HEYMER, S., AND TRETMANS, J. 1996. Report on the standardization project "formal methods in conformance testing". In *Proceedings of the IFIP TC6 9th international workshop on Testing of communicating systems*, B. Baumgarten, H.-J. Burkhardt, and A. Giessler, Eds. Chapman & Hall, London, 289–298.

HONG, H. S., LEE, I., SOKOLSKY, O., AND URAL, H. 2002. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 2280. Springer, Berlin / Heidelberg, Germany, 151–161.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, USA.

HUO, J. AND PETRENKO, A. 2004. On testing partially specified iots through lossless queues. In *Proceedings of the 16th IFIP International Conference Testing of Communicating Systems*, R. Groz and R. M. Hierons, Eds. Lecture Notes in Computer Science, vol. 2978. Springer, Berlin / Heidelberg, Germany, 76–94.

HUO, J. AND PETRENKO, A. 2005. Covering transitions of concurrent systems through queues. In *Proceedings of the 16th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 335–345.

HUO, J. AND PETRENKO, A. 2009. Transition covering tests for systems with queues. *Software Testing, Verification and Reliability 19,* 1 (March), 55–83.

IEEE. 1990. IEEE standard glossary of software engineering terminology. Std 610.121990.

ISO. 1989. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. www.iso.org.

ISO. 1994. ISO/IEC 9646-1: Information technology – open systems interconnection – conformance testing methodology and framework – part 1: General concepts. www.iso.org.

ISO. 1997. Information retreival, transfer, and management for osi. framework: Formal methods in conformance testing. ISO/ICE JTC1/SC21 WG7, ITU-T SG 10/Q.8, Committee Draft CD 13245-1, ITU-T Proposed Recommendation Z.500. Geneve, Switzerland.

ISO. 2002. ISO/IEC 13568: Information technology – z formal specification notation – syntax, type system and semantics. www.iso.org.

ITU-T. 1998. Formal description techniques (FDT) - message sequence chart (MSC). ITU-T Recommendation Z.120 - Annex B.

JARD, C. 2003. Synthesis of distributed testers from true-concurrency models of reactive systems. *Information & Software Technology 45,* 12, 805–814.

JARD, C. AND JÉRON, T. 2005. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer 7,* 4 (August), 297–315.

JARD, C., JÉRON, T., KAHLOUCHE, H., AND VIHO, C. 1998. Towards automatic distribution of testers for distributed conformance testing. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification,Testing and Verification*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 353–368.

JARD, C., JÉRON, T., TANGUY, L., AND VIHO, C. 1999. Remote testing can be as powerful as local testing. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, J. Wu, S. Chanson, and Q. Gao, Eds. Kluwer Academic Publishers, Dordrecht, The Netherlands, 25–40.

JEANNET, B., JÉRON, T., RUSU, V., AND ZINOVIEVA, E. 2005. Symbolic test selection based on approximate analysis. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, N. Halbwachs and L. D. Zuck, Eds. Lecture Notes in Computer Science, vol. 3440. Springer, Berlin / Heidelberg, Germany, 349–364.

JONES, C. B. 1990. *Systematic Software Development Using VDM*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, USA.

JONES, J. A., HARROLD, M. J., AND STASKO, J. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, New York, NY, USA, 467–477.

JÜRJENS, J. 2008. Model-based security testing using umlsec: A case study. *Electronic Notes in Theoretical Computer Science 220,* 1 (December), 93–104.

KAHLOUCHE, H., VIHO, C., AND ZENDRI, M. 1999. Hardware testing using a communication protocol conformance testing tool. In *Proceedings of the 5th International Conference Tools and Algorithms for Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 1579. Springer, Berlin / Heidelberg, Germany, 315–329.

KANER, C., BACH, J., AND PETTICHORD, B. 2001. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

KHOUMSI, A., JÉRON, T., AND MARCHAND, H. 2004. Test cases generation for nondeterministic real-time systems. In *Proceedings of the 3rd International Workshop on Formal Approaches to Software Testing*, A. Petrenko and A. Ulrich, Eds. Lecture Notes in Computer Science, vol. 2931. Springer, Berlin / Heidelberg, Germany, 131–146.

KING, J. C. 1975. A new approach to program testing. *ACM SIGPLAN Notices 10,* 6, 228–233.

KRICHEN, M. AND TRIPAKIS, S. 2004. Black-box conformance testing for real-time systems. In *Proceedings of the 11th International SPIN Workshop*, S. Graf and L. Mounier, Eds. Lecture Notes in Computer Science, vol. 2989. Springer, Berlin / Heidelberg, Germany, 109–126.

LEDRU, Y., DU BOUSQUET, L., BONTRON, P., MAURY, O., ORIAT, C., AND POTET, M.-L. 2001. Test purposes: adapting the notion of specification to testing. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*. IEEE Computer Society, Washington, DC, USA, 127–134.

LEE, D. AND YANNAKAKIS, M. 1996. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE 84,* 8 (August), 1090–1123.

LESTIENNES, G. AND GAUDEL, M.-C. 2002. Testing processes from formal specifications with inputs, outputs and data types. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*. 3–14.

LESTIENNES, G. AND GAUDEL, M.-C. 2005. Test de systèmes réactifs non réceptifs. *Journal Européen des Systèmes Automatisés, Modélisation des Systèmes Réactifs 39,* 1–3, 255–270. in French, Technical Report in English available.

LI, Z., WU, J., AND YIN, X. 2003. Refusal testing for miots with nonlockable output channels. In *Proceedings of the International Conference on Computer Networks and Mobile Computing*. IEEE, Washington, DC, USA, 517–522.

LI, Z., WU, J., AND YIN, X. 2004. Testing multi input/output transition system with all-observer. In *Proceedings of the 16th IFIP International Conference on Testing Communication Systems*, R. Groz and R. M. Hierons, Eds. Lecture Notes in Computer Science, vol. 2978. Springer, Berlin / Heidelberg, Germany, 95–111.

LI, Z., YIN, X., AND WU, J. 2004. Distributed testing of multi input/output transition system. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*. IEEE, Washington, DC, USA, 271–280.

LUO, G., VON BOCHMANN, G., AND PETRENKO, A. 1994. Test selection based on communicating nondeterministic finite-statemachines using a generalized wp-method. *Transactions on Software Engineering 20,* 2, 149–162.

MANNA, Z. AND PNUELI, A. 1995. *Temporal verification of reactive systems: safety*. Springer-Verlag, Inc, New York, NY, USA.

MATEESCU, R. AND OUDOT, E. 2008. Bisimulator 2.0: An on-the-fly equivalence checker based on boolean equation systems. In *Proceedings of the 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*. IEEE Computer Society, Washington, DC, USA, 73–74.

MATEESCU, R. AND SIGHIREANU, M. 2000. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming - Special issue on formal methods for industrial critical systems 46,* 3, 255–281.

MILNER, R. 1980. *A Calculus of Communicating Systems*. Vol. 92. Springer, Berlin / Heidelberg, Germany.

MILNER, R. 1989. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

MILNER, R. 1990. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, J. van Leeuwen, Ed. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, Chapter 19, 1201–1242.

MYERS, G. J. 1979. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA. Revised and updated in 2004 by Tom Badgett and Todd M. Thomas with Corey Sandler.

OFFUTT, A. J. AND VOAS, J. M. 1996. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-01, ISSE Department of George Mason University, Fairfax, VA 22030. January.

OKUN, V., BLACK, P. E., AND YESHA, Y. 2002. Testing with model checker: Insuring fault visibility. In *Proceedings of the International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering System*. 1351–1356.

OLIVEIRA, M. V. M., CAVALCANTI, A. L. C., AND WOODCOCK, J. C. P. 2007. A UTP semantics for Circus. *Formal Aspects of Computing 21,* 1, 3 – 32.

OWRE, S., RUSHBY, J. M., AND SHANKAR, N. 1992. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, D. Kapur, Ed. Lecture Notes in Artificial Intelligence, vol. 607. Springer-Verlag, Saratoga, NY, 748–752.

PEISCHL, B., WEIGLHOFER, M., AND WOTAWA, F. 2007. Executing abstract test cases. In *Proceedings of the Model-based Testing Workshop in conjunction with the 37th Annual Congress of the Gesellschaft fuer Informatik*. GI, Germany, 421–426.

PETRENKO, A., BORODAY, S., AND GROZ, R. 2004. Confirming configurations in efsm testing. *IEEE Transactions on Software Engineering 30,* 1, 29–42.

PETRENKO, A. AND YEVTUSHENKO, N. 2002. Queued testing of transition systems with inputs and outputs. In *Proceedings of the Workshop Formal Approaches to Testing of Software*, R. Hierons and T. Jéron, Eds. 79–93.

PETRENKO, A. AND YEVTUSHENKO, N. 2005. Conformance tests as checking experiments for partial nondeterministic fsm. In *Proceedings of the 5th International Workshop on Formal Approaches to Software Testing*. Lecture Notes in Computer Science, vol. 3997. Springer, Berlin / Heidelberg, Germany, 118–133.

PETRENKO, A., YEVTUSHENKO, N., AND HUO, J. L. 2003. Testing transition systems with input and output testers. In *Proceedings of the 15th IFIP International Conference on Testing of Communication Systems*, D. Hogrefe and A. Wiles, Eds. Lecture Notes in Computer Science, vol. 2644. Springer, Berlin / Heidelberg, Germany, 129–145.

PHILIPPS, J., PRETSCHNER, A., SLOTOSCH, O., AIGLSTORFER, E., KRIEBEL, S., AND SCHOLL, K. 2003. Model-based test case generation for smart cards. *Electronic Notes in Theoretical Computer Science 80*, 1–15.

PHILLIPS, I. 1987. Refusal testing. *Theoretical Computer Science 50,* 3, 241–284.

PRENNINGER, W., EL-RAMLY, M., AND HORSTMANN, M. 2005. *Case Studies*, Chapter 15, 439–461, of Broy et al. (2005).

PRENNINGER, W. AND PRETSCHNER, A. 2005. Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science 116*, 59–71.

PRETSCHNER, A. AND PHILIPPS, J. 2005. *Methodological Issues in Model-Based Testing*, Chapter 10, 281–291, of Broy et al. (2005).

PRETSCHNER, A., PRENNINGER, W., WAGNER, S., KÜHNEL, C., BAUMGARTNER, M., SOSTAWA, B., ZÖLCH, R., AND STAUNER, T. 2005. One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, St. Louis, Missouri, USA, 392–401.

PYHÄLÄ, T. AND HELJANKO, K. 2003. Specification coverage aided test selection. In *Proceedings of the 3rd International Conference on Application of Concurrency to System Design*. IEEE Computer Society, Washington, DC, USA, 187–195.

RAYADURGAM, S. AND HEIMDAHL, M. P. E. 2001. Coverage based test-case generation using model checkers. In *Proceedings of the 8th International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE Computer Society, Washington, DC, USA, 83–91.

REPS, T., BALL, T., DAS, M., AND LARUS, J. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering conference*. Springer-Verlag, Inc., New York, NY, USA, 432–449.

ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLEY, M., AND SCHOOLER, E. 2002. SIP: Session initiation protocol. RFC 3261, IETF.

ROTHERMEL, G., HARROLD, M. J., VON RONNE, J., AND HONG, C. 2002. Empirical studies of test suite reduction. *Journal of Software Testing, Verification, and Reliability 4,* 2 (December), 219–249.

RUSU, V., DU BOUSQUET, L., AND JÉRON, T. 2000. An approach to symbolic test generation. In *Proceedings of the 2nd International Conference on Integrated Formal Methods*, W. Grieskamp, T. Santen, and B. Stoddart, Eds. Lecture Notes in Computer Science, vol. 1945. Springer, Berlin / Heidelberg, Germany, 338–357.

SCHMITT, M., EK, A., KOCH, B., GRABOWSKI, J., AND HOGREFE, D. 1998. Autolink - putting sdl-based test generation into practice. In *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*. Kluwer, B.V., Deventer, The Netherlands, 227–244.

SEGALA, R. 1997. Quiescence, fairness, testing, and the notion of implementation. *Information and Computation 138,* 2, 194–210.

SRIVATANAKUL, T., CLARK, J. A., STEPNEY, S., AND POLACK, F. 2003. Challenging formal specifications by mutation: a csp security example. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*. IEEE, Washington, DC, USA, 340–350.

TAN, Q. M. AND PETRENKO, A. 1998. Test generation for specifications modeled by input/output automata. In *Proceedings of the 11th International Workshop on Testing Communicating Systems*. IFIP Conference Proceedings, vol. 131. Kluwer Academic Publishers, Dordrecht, The Netherlands, 83–100.

TANGUY, L., VIHO, C., AND JARD, C. 2000. Synthesizing coordination procedures for distributed testing of distributed systems. In *Proceedings of the Workshop on Distributed System Validation and Verification*. E67–E74.

TARSKI, A. 1941. On the calculus of relations. *Journal of Symbolic Logic 6,* 3, 73–89.

TERPSTRA, R., PIRES, L. F., HEERINK, L., AND TRETMANS, J. 1996. Testing theory in practice: A simple experiment. Tech. rep., University of Twente, The Netherlands.

TIP, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3.

TRETMANS, J. 1992. A formal approach to conformance testing. Ph.D. thesis, University of Twente, Enschede.

TRETMANS, J. 1996. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools 17,* 3, 103–120.

TRETMANS, J. 1999. Testing concurrent systems: A formal approach. In *Proceedings of the 10th International Conference on Concurrency Theory*. Vol. 1664. Springer, Berlin / Heidelberg, Germany, 779–799.

TRETMANS, J. 2008. *Model Based Testing with Labelled Transition Systems*, 1–38, of Hierons et al. (2008).

TRETMANS, J. AND BRINKSMA, E. 2003. TorX: Automated model based testing. In *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Zieger, Eds. AGEDIS project, Nurnburg, Germany, 13–25.

TURNER, K. J. 1989. The formal specification language LOTOS - a course for users. Tech. rep., University of Stirling - Department of Computing Science and Mathematics, Stirling, Scotland. August.

TURNER, K. J. 1993. *Using Formal Description Techniques - An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., Hoboken, NJ, USA.

ULRICH, A. AND KÖNIG, H. 1997. Specification-based testing of concurrent systems. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*. IFIP Conference Proceedings, vol. 107. Chapman & Hall, London, UK, 7–22.

UTTING, M. AND LEGEARD, B. 2007. *Practical Model-Based Testing. A Tools Approach.* Morgan Kaufmann, San Francisco, CA, USA.

VAN DER BIJL, M., RENSINK, A., AND TRETMANS, J. 2003. Compositional testing with ioco. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software*. Lecture Notes in Computer Science, vol. 2931. Springer, Berlin / Heidelberg, Germany, 86–100.

VAN DER BIJL, M., RENSINK, A., AND TRETMANS, J. 2005. Action refinement in conformance testing. In *Proceedings of the 17th IFIP International Conference on Testing of Communicating Systems*, K. Ferhat and D. Rachida, Eds. Lecture Notes in Computer Science, vol. 3502. Springer, Berlin / Heidelberg, Germany, 81–96.

VAN DER SCHOOT, H. AND URAL, H. 1995. Data flow oriented test selection for lotos. *Computer Networks and ISDN Systems 27,* 7, 1111–1136.

VAN OSCH, M. 2006. Hybrid input-output conformance and test generation. In *Proceedings of the 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*, K. Havelund, M. Núñez, G. Rosu, and B. Wolff, Eds. Lecture Notes in Computer Science, vol. 4262. Springer, Berlin / Heidelberg, Germany, 70–84.

VEANES, M., CAMPBELL, C., GRIESKAMP, W., SCHULTE, W., TILLMANN, N., AND NACHMANSON, L. 2008. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, 39–76, of Hierons et al. (2008).

VERHAARD, L., TRETMANS, J., KARS, P., AND BRINKSMA, E. 1993. On asynchronous testing. In *Proceedings of the 5th International Workshop on Protocol Test Systems*. IFIP Transactions. North-Holland Publishing Co., Amsterdam, The Netherlands, 55–66.

VIHO, C. 2005. Test distribution: a solution for complex network system testing. *International Journal on Software Tools for Technology Transfer 7,* 4 (August), 316–325.

WEIGLHOFER, M. 2006. A LOTOS formalization of SIP. Tech. Rep. SNA-TR-2006-1P1, Competence Network Softnet Austria, Graz, Austria. December.

WEIGLHOFER, M. AND AICHERNIG, B. K. 2008a. Input output conformance testing in the unifying theories of programming. Tech. Rep. SNA-TR-2008-1P6, Competence Network Softnet Austria. online: http://www.ist.tugraz.at/staff/weiglhofer/publications.

WEIGLHOFER, M. AND AICHERNIG, B. K. 2008b. Unifying input output conformance. In *Proceedings of the 2nd International Symposium on Unifying Theories of Programming*, A. Butterfield, Ed. Lecture Notes in Computer Science, vol. 5713. Springer, Berlin / Heidelberg, Germany.

WEIGLHOFER, M., AICHERNIG, B. K., AND WOTAWA, F. 2009. Fault-based conformance testing in practice. *International Journal of Software and Informatics 3,* 2-3 (September), 375–411.

WEIGLHOFER, M., FRASER, G., AND WOTAWA, F. 2009a. Using coverage to automate and improve test purpose based testing. *Information and Software Technology 51,* 11 (November), 1601—-1617.

WEIGLHOFER, M., FRASER, G., AND WOTAWA, F. 2009b. Using spectrum-based fault localization for test case grouping. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computing Society, Washington, DC, USA.

WEIGLHOFER, M. AND WOTAWA, F. 2008a. "On the fly" input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*. ACTA Press, Calgary, Canada, 286–291.

WEIGLHOFER, M. AND WOTAWA, F. 2008b. Random vs. scenario-based vs. fault-based testing: An industrial evaluation of formal black-box testing methods. In *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering*. INSTCC Press, Portugal, 115–122.

WEIGLHOFER, M. AND WOTAWA, F. 2009a. Asynchronous input-output conformance testing. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*. IEEE Computer Society, Washington, DC, USA, 154–159.

WEIGLHOFER, M. AND WOTAWA, F. 2009b. Improving coverage based test purposes. In *Proceedings of the 9th International Conference on Quality Software*. IEEE Computer Society, Washington, DC, USA. to appear.

WEISER, M. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, Piscataway, NJ, USA, 439–449.

WEST, C. H. 1989. Protocol validation in complex systems. *ACM SIGCOMM Computer Communication Review 19,* 4, 303–312.

WHITTAKER, J. A. 2000. What is software testing? and why is it so hard? *IEEE Software 17,* 1 (January/February), 70–79.

WOODCOCK, J. AND CAVALCANTI, A. 2004. A tutorial introduction to designs in unifying theories of programming. In *Proceedings of 4th International Conference on Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds. Lecture Notes in Computer Science, vol. 2999. Springer, Berlin / Heidelberg, Germany, 40–66.

ZOETEWEIJ, P., ABREU, R., GOLSTEIJN, R., AND VAN GEMUND, A. 2007. Diagnosis of embedded software using program spectra. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE Computer Society, Washington, DC, USA.