On changing models in Model-Based Testing

Machiel van der Bijl

CTIT Dissertation Series No. 11-198
Center for Telematics and Information Technology (CTIT)
P.O. Box 217 - 7500AE Enschede - the Netherlands
ISSN: 1381-3617

IPA Dissertation Series No. 2011-07
The work in this thesis has been carried out under the auspices of the Institute for Programming Research and Algorithms (IPA) research school.

This research was supported by the Dutch research program PROGRESS under project: TES5417: Atomyste ATOm splitting in eMbedded sYStems TEsting.

ON CHANGING MODELS IN MODEL-BASED TESTING

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 12 mei 2011 om 12.45 uur

door

Hendrik Michaël van der Bijl

geboren op 10-08-1973
te Dordrecht

ISBN: 978-90-365-3195-5

Acknowledgments

Doing a Ph.D. is an interesting phenomenon that has a profound impact on the researcher and his environment. I am very thankful that I have been given this opportunity (and that I have taken it). I am indebted to a long list of people that supported me through the years, too many to name them all, but I thank you all.

Many thanks are due to Ed Brinksma, Arend Rensink and Jan Tretmans. For sharing numerous insights in commenting upon my work, for teaching me the tricks of the trade and for keeping up with me. You taught me a lot and I thank you!

I am grateful to the members of my graduation committee for spending their precious time and for their useful comments.

My colleagues of the FMT group at the Universiteit Twente. You have made my days into a most enjoyable and fruitful experience. In particular I would like to thank my roomies Joost Noppen and Bedir Tekinerdogan for pleasant times and interesting, lively discussions.

I also want to thank Kate Demuth and Mark Johnson. Thanks for showing me the wonderful world of science and piquing my interest with your enthusiasm. Thanks for broadening my horizons.

Many thanks to Pim Kars for introducing me to the fascinating area of software testing, for showing me the formal world of software construction and for many pleasant conversations. You showed me that one can mix business with science and pleasure (or have I got it all backwards?).

My dear colleagues at Axini, thanks for your patience while my *boekje* was not yet, or almost, finished. I took my time and I thank you for your support, teasing and lots of fun at work.

Finally, I would like to thank all my friends and family for their encouragement and moral support over the years. In particular I would like to thank my parents, without their love, support and care it would not have happened. Last but not least I thank Rudina, for the things that really matter.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SOME TWO DECADES AGO, during my studies, I[1] came into contact with the world of software testing for the first time. I was working on a piece of software that was bigger than anything I had written before and it was showing faulty behavior. How annoying! During my studies I had learned a bit about software testing. However in practice there were not many tools to help me, except for the debugger. With hard hard work, smart thinking and the use of some home grown testing/debugging tools I managed to solve my problems. Now, with some years of experience in the software development world, I know that software testing is an extremely difficult problem begging to be solved. One might even argue that the problem of recognizing a correctly behaving system is at least as difficult as, if not more difficult than building the system itself. The difficulty of software testing and the fact that the software development world does not realize and/or recognize this, is one of the main motivations for doing this research.

1.1 Why software testing is difficult

In this thesis we use a rather liberal notion of software: the instructions for a machine to perform a certain functionality automatically. In general, with instructions we mean a computer program written in some kind of programming language and with machine we mean a computer, also known as hardware. With "certain functionality", we mean the functionality that the program or machine was made for, for example electronic banking for an electronic banking program, text editing for a word processor, dispensing cash for an ATM, etc. Many machines these days have a computer inside that runs software, for example the computer we use daily to check our email and our cell-phone. Less trivial examples are television sets, cars and electronic razors.

[1]This introduction sometimes reflects the personal opinions of the author. In those cases we take the liberty to use the first-person singular.

With software testing we mean the activity of experimenting with the software in order to find out if the software, or the software in combination with the hardware, is functioning the way it is intended to do.

Testing software is an important activity, because it inspects the quality of the software. There are also other techniques to inspect and improve the quality of software, for example model-checking and theorem proving, but testing is the technique most used in practice.

Testing software is difficult for at least two reasons. One is that it is not always clear what the required functionality of the software is. The other is the size and complexity of the software itself. To test a big part of the intended functionality of a software system one may have to perform thousands of tests, which easily takes several man weeks or man months. There are two main reasons for the hugeness, one is the amount of possible data combinations in a system and the other one is the amount of possible interactions with the system. We illustrate these problems in the following examples.

Example 1.1.1 Suppose we want to thoroughly test the addition function of a calculator. Let us simplify this by only adding two numbers and by only using integer values (whole numbers, i.e., no fractions) with only 8 digits. And we simplify this test even more by only using positive numbers. This means that we can choose numbers from 0 to 99999999, in other words 10^8 possible options. This means that there are 10^{16} combinations for the two numbers. Suppose that every test takes 5 micro-seconds, this means that completely testing the function takes around 20294 years (note that we are only taking the test execution into account and leave out the creation of the test-case and checking the outcome of the test). I think we all agree that this is a lot of test-cases and a lot of time for only this small part of the total functionality. □

This may seem a trivial example. And what is the big deal with calculators, we all know that they work, right? Well, it took quite a while before calculators worked with nowadays perfection, and also these days we sometimes run into problems that could have been detected, had we been able to test more thoroughly. Remember the Pentium bug [Wol94] and the Ariane 5 software bug [Nus97]? And also do not forget that there are indeed more complex systems, these are only integer calculations. Translate this “simple” case to more complex software, for example the administration system of a pension provider, or the software for space exploration, like the Mars Pathfinder.

The other issue with software testing is the amount of interactions that are possible with software systems.

Example 1.1.2 Let us stick with our calculator example. Most calculators have the possibility to correct wrong inputs by pressing a correction key;

quite often labeled ‘C’. This works as follows, when I want to enter the number ‘10’, but by accident I start by pressing the number ‘2’, I can correct this by pressing the ‘C’ button. The result is that the entered number resets to ‘0’ and I can try again to enter the number. The effect is that there are basically an infinite amount of possibilities to enter every number. For example, when we want to enter the number ‘1’, we can press ‘1’, but we can also press ‘2’, followed by ‘C’, followed by ‘1’, or ‘2’, followed by ‘C’, followed by ‘1’, followed by ‘C’, followed by ‘1’, or . . .

This phenomenon does not only occur with calculators, but for example also with web-based applications, where one can use the “back”-button to go back to the previous page, or for example the back-button on your navigation system. □

The correction key is an example that increases the number of possible interactions with a system dramatically. But even when we disregard the correction keys and back-buttons of this world, nowadays systems are quite complex to test. Take for example OpenOffice 2.4, an open source word processor. On a quick inspection we find at least 139 menu items, some of which have sub-menus and some of which can be combined with other options.

With bigger and more complex systems the number of tests that we need to test the system only gets bigger. In other words, the problem that we face with software testing is that we want to assure the quality of a practically infinite system in a finite amount of time. By hand, it is simply impossible. Testing a relevant part of the system under test, simply calls for massive automation of the entire test process. And even then we won’t come close to the amount of tests that we theoretically need to execute. There are some tools to automate test-case execution, but these still require the test-cases to be written by hand. Examples are QTP by HP and Rational Robot by IBM (see their respective websites, www.hp.com and www.ibm.com, for details, this information changes from day to day). If only we could automate the test-case generation part, this would mean that we could test a significantly bigger part of the software.

A question often heard is whether we really need these big numbers in practice? Well, yes and no. No, in practice we do not need these billions of test-cases. It is impractical to execute them anyway, because we do not have enough time. But more importantly, in most software we use only a core set of functionality most of the time. We do not even remotely use all the possible interactions with the system and neither do we use all possible data combinations. Furthermore, most software is used for a finite amount of time, for example, because the system is reset periodically. This also limits the amount of possible test-cases. On the other hand the answer is yes. The amounts of tests that we perform manually (in the tens or hundreds) are by far not enough. Yes, we really need thousands of test-cases to thoroughly

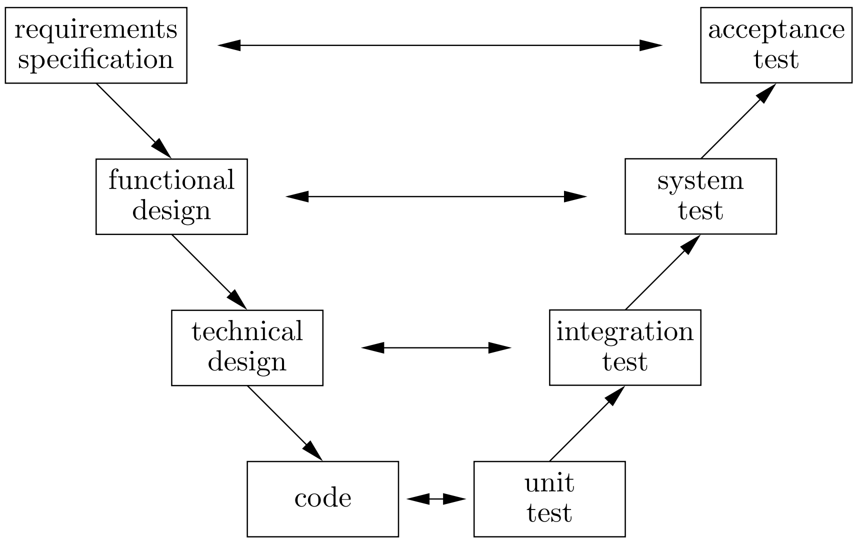


Figure 1.1: V-Model

test a software system. The other day I saw at a client that our software [Axi] found an error after 90.000 tests. So yes, practice shows that we really need these big numbers.

On top of this, practice is that testers are at the bottom of the food chain in the software development world: bad programmers become testers, new hires start as testers, people with backgrounds that are not even remotely related to computer science become testers. On top of the complexity of testing, it does not help if testing is done by people not suited for the job. Furthermore, in general software is not developed with testing in mind. Let us take a bird's eye view at how software is developed in practice.

1.1.1 Software development in the real world

I will quickly sketch how software is developed by using the so called "V-model" in Figure 1.1. Note that my description here of the way how software is developed is a gross oversimplification and at the same time a rather accurate account of the way software is developed in practice (based on some years of experience). For those interested in a more detailed description we refer to [Pre04]. On the left-hand side of Figure 1.1 we see design activities and on the right-hand side we see corresponding test activities. In general, software development is started with a phase where domain specialists (for example insurance specialists when building an insurance administration system) talk with the customer, or the intended users of the software system. These domain specialists write down what the system should do, the *what*, from a perspective of the client. The document that captures the requirements of the system is generally called "requirements specification". Based on the requirements specification, software designers make a "functional design" that describes the system to be built from a functional perspective; the *how*. Next there is a phase in which software engineers translate the requirements and functional design into a technical description of the required software system; the "technical design". The next phase is the actual building of the system: the writing of code in a particular programming language or environment for a specific platform. For bigger software systems this means that the functionality is split up in smaller parts and distributed over several programmers. The end of this phase should be the delivery of the actual software system that the client requested. That is easier said than done. How do we recognize that the realized system is the one that the customer wanted? There are several ways to do this, but the most used technique in practice is *testing*. Testing means executing and using the software system and (manually) checking if the system behaves as expected. When this is done in a structured way, several tests are performed, as we can see on the right-hand side of Figure 1.1 on the facing page: *acceptance test*, *system test*, *integration test* and *unit test*. Best case, these tests are made during system design, based on the available system documentation, so that they can be executed when (parts of) the system are ready. Experience shows that software projects are particularly bad at staying on schedule. By the time the software is ready to be tested there is almost no time left to execute the test-cases, let alone prepare them. We describe the tests, starting at the bottom with unit-tests and working our way up in the V-model. Unit tests are used to check the parts (also known as units, hence the name) of the system that the programmers made. In general these tests are done by the programmer. When several units are ready we can integrate them and check if they behave according to the "technical design". When the system is deemed to be correct from a technical perspective, a system test is performed to check if the system behaves as described by the the "functional design". Last but not least the system is tested with or by the customer to see if it complies with the original "requirements", the so called acceptance test.

In my experience as a software engineer, doing projects for financial organizations, government and companies in the embedded system world, most software testing is done manually: test-cases are written by hand, they are executed by hand and the outcome of the test is checked by hand. The good part about this is that if the manual testing is done the right way, in accordance with the software development activities it can result in decent quality software. The downside is that it takes a lot of time and effort. Apart from the time to execute all the tests once, it often happens that tests need to be rerun several times. This happens for example when an error is corrected in a new release of the software and the fix needs to be re-tested. Or when new functionality is added to the software and we retest the existing functionality, to ensure that it is not negatively effected

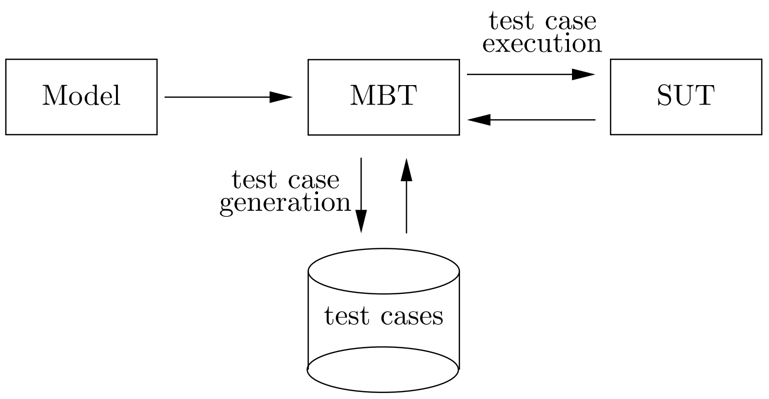


Figure 1.2: Model-Based Testing

by the new functionality. It is not uncommon that the same test needs to be re-run between ten and twenty times for one release of the software. Even with these “good” projects it often happens that serious bugs remain in the software, simply because manual testing cannot cover enough of the functionality of the system under test.

An interesting candidate testing solution to test with more test-cases is Model-Based Testing (MBT). MBT is different from other approaches because it can automatically generate test-cases. This is important, because it makes it possible to come up with a significantly bigger set of test-cases than possible by manual testing. The promise of MBT is not only that we test the software more thoroughly, but also that we are able to test quicker, repeatable and in a more flexible fashion. This makes it possible to give software developers quick and thorough feedback, enabling them to shorten their development cycle. This means that they can make better quality software in shorter time.

1.2 Model-Based Testing

In Chapter 2 we introduce MBT in more detail, but in short MBT works as follows (see Figure 1.2). The basis is a model, this is a functional description of the software system that we want to test (also known as System Under Test, or SUT), similar to, but more structured than the requirements specification and the functional design. Because the model is written using a formal written notation, we can analyze it and derive test-cases from it. We can store these test-cases in a database and execute them against the system under test. Because the model describes the functionality of the system under test we also know the allowed responses of the system to the test-case, hence we can also automatically evaluate the outcome of the test.

The derived test-cases test if the software system complies with the functionality as defined in the model. Here of course lies an interesting aspect of MBT. How do we know whether we have a correct model; quis custodiet ipsos custodes? Note that this problem is not new, it also exists with manual testing: how do we know whether a test-case is correct? In the traditional setting we have documentation and domain experts as the basis for our test-cases. With MBT we have some extra possibilities. We have the model itself that can be reviewed by domain experts. With the right formal underpinning, the model is executable. Therefore we can simulate its behavior. Furthermore there is also other research that focuses on these questions, for example research in model checking [BK08]. In this thesis we assume that we have a correct model. We focus on the situation that eventually the model and/or the test-cases will change. This is not a problem, but a fact of life. It is also our experience when applying model-based testing in practice. Models and test-cases change, for example when we find differences in granularity between the model and the system under test. Actions in the model are implemented slightly differently in the system under test than what was described in the specification, or another possibility is that in the model we abstracted from functionality that turns out to be necessary to test the system. In this thesis we look at ways to use and change models and test-cases in a flexible way. Our research questions are centered around *modular design* and *action refinement*.

1.3 Research questions

Modular design means that we split the functionality of the entire system into smaller coherent parts. Because these parts are smaller, they are easier to model, to maintain and to test. This technique, also known as “divide and conquer”, is a well known engineering technique. Think for example of the way an automobile is split up into coherent parts: the engine, the body work, the suspension, etc. We investigate modular design and modular testing for MBT in Chapter 3. The research question treated in this chapter is:

- Given that components (individually) have been tested and found correct, may we conclude that their integrated behavior is also correct? If this is the case it would imply that we only have to test the parts of a system and not the system as a whole!

In system modeling, modular design has been investigated extensively, for example in several process algebraic formalisms, but in MBT this has not been the case. We illustrate testing modular design, also known as *component based testing*, in the following example. As is the tradition in model-based testing at the University of Twente, the example is a coffee-machine.

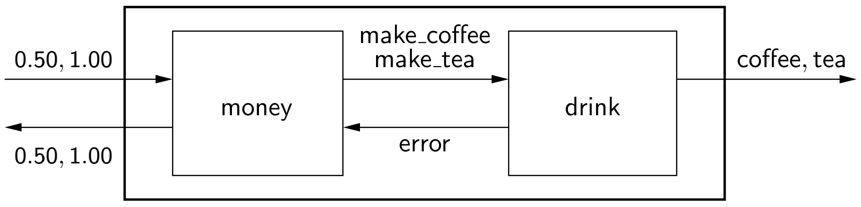


Figure 1.3: Schematic overview of a coffee machine

Example 1.3.1 In Figure 1.3 we schematically show a coffee-machine. The specification of the machine is extremely simple, it works as follows: when we enter 50 cent we get a cup of tea and when we enter 1 euro we get a cup of coffee. When something goes wrong in the drink making process we get our money back. The machine consists of two parts, a part that takes care of the money and a part that takes care of the drinks. When the money component receives 50 cent, it gives a *make_tea* command to the drink component. And after receiving the command, the drink component delivers *tea*. Likewise it produces *coffee* after the *make_coffee* command. When something goes wrong in the drink component it gives an *error* signal to the money component, which gives the inserted money back. □

The million dollar question is: if we test the money and drink components completely and find them correct, does that mean that the entire coffee machine is correct? The shortest answer is *no*, the longer answer is *yes* under certain conditions. This longer answer is treated in Chapter 3.

Modular design makes it easier to create and maintain models but at a certain moment models change, or the test-cases that were generated from these models change. Is there a way to keep the models and test-cases aligned? We could of course make the changes by hand, but this often turns out to be an error prone and laborious exercise. More importantly, we want to be able to change already derived test suites in such a way that they are still correct with respect to the changed model. This means that we have to change the model as well as the test-cases. If possible, we would like to do this automatically in a controlled manner. Model transformation is a technique that makes it possible to change behavior of a system in an automatic way by adding or removing functionality. An interesting model transformation technique for MBT is *action refinement* [GR01]. This technique has been studied in model design but it is unclear how action refinement works for MBT. Especially it is unclear how to apply action refinement to test-cases. We study action refinement for model-based testing in Chapter 5. The research questions treated in this chapter are:

- How can we refine models and test-cases with inputs and outputs? The theories found in the literature do not make this distinction.

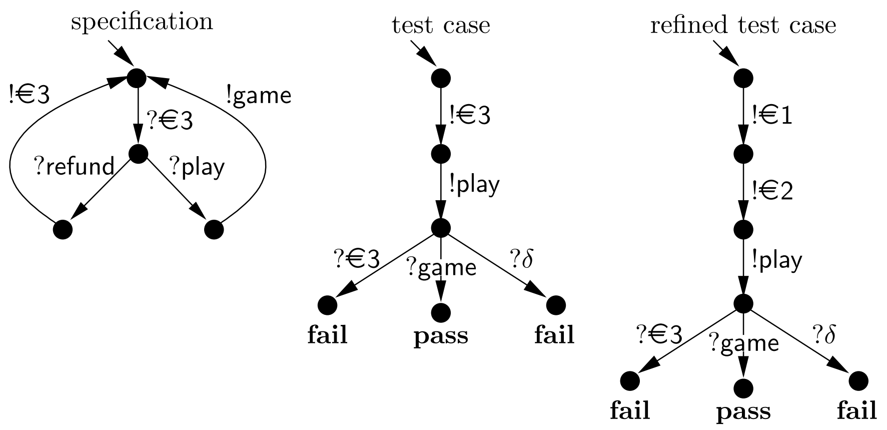


Figure 1.4: Video game specification with test-cases

- Suppose we refine a set of test-cases that is derived from a model. Likewise we refine the model and generate a set of test-cases. What can we say about the relation between the set of refined test-cases and the set of test-cases derived from the refined model?

Example 1.3.2 On the left-hand side in Figure 1.4 we show the state machine of a video game. The black dots are states, the start state has a short incoming arrow that is not connected to another state, the arrows denote transitions. Question marks denote input actions and exclamation marks output actions. Together this reads as follows: we enter 3 euro and then we may press the *play* button and play the game, or we may press the *refund* button to get our money back. With MBT we can automatically generate test-cases from this specification, for example the one in the middle of Figure 1.4. This one reads: enter 3 euro, press the *play* button and make an observation (this is the fork in the tree). The only correct answer is the observation of the *game* (pass). The observation of 3 euro, or nothing (represented by the symbol δ) leads to a fail verdict.

The thing is, there do not exist 3 euro coins. In other words we need to make it more explicit what we mean with 3 euro. Suppose for example, that with 3 euro we mean: 1 euro followed by 2 euro or 2 euro followed by 1 euro. With this information, we want to refine (read enhance) our test-case to the test-case shown on the right-hand side of Figure 1.4. This test-case reads: after entering 1 euro followed by 2 euro and pressing the *play* key, only *game* is a correct response of the system. Other responses lead to a fail verdict. □

1.4 Overview of the thesis

This thesis is part of the research in model-based testing. In Chapter 2 we introduce model-based testing. In Chapter 3 we show under which restrictions modular design works for model-based testing. Action refinement in model-based testing is introduced in Chapter 4. Here we explain what action refinement is and what problem we hope to solve by applying action refinement to model-based testing. In Chapter 5 we present our action refinement theory for model-based testing. We end with our conclusions in Chapter 6.

Chapter 2

Model-Based Testing

In this chapter we make clear what we mean by model-based testing and we introduce the **ioco** test theory that we use, including test-case generation and execution.

2.1 Introduction

Most of the "real world" testing of software systems is done by hand. Testers specify test-cases by hand, execute them by hand and evaluate the outcome by hand. They get the necessary knowledge for the test-cases by reading system documentation, by talking to future users and designers of the system and by using their experience. One could say that the quality of the tests depends mainly on the skill of the tester.

Model-Based Testing (MBT) aims at automating the process of *specifying* and *executing* test-cases, and *evaluating* the outcome of the test execution. A unique property of MBT compared to other test approaches is that it enables the automatic generation of test-cases. Central in the approach is that the desired system functionality[1] is specified in a *formal*, i.e., mathematical model. From a practical perspective, a model is formal enough if it can be manipulated automatically in order to construct test-cases; in this sense a computer program may be a formal model. MBT uses a notion of correctness (also known as implementation relation) together with the information in the model to derive test-cases. This means that with MBT the quality of the test-cases depends on the quality of the model and the quality of the algorithm to derive test-cases from the model. In the world of MBT, or more specifically the realm of testing reactive systems, there are basically two formalisms used, those based on Finite State Machines (FSM) and those based on Labeled Transition Systems (LTS). FSM based testing

[1]We use the term *functionality* here in a broad sense: "the desired behavior of the system", this may include so-called extra-functional behavior like performance, security, etc.

has a long tradition: already in 1956, Moore wrote a seminal paper about it [Moo56] in which he introduced the idea of experimenting with an FSM to draw conclusions about its internal state. For an annotated bibliography on FSM testing see [Pet00], for more information see [BJK$^+$05, LY96]. The formal testing theory on which LTS based testing is based was introduced by De Nicola and Hennessy in 1984 [DNH84]. For an annotated bibliography on LTS based testing see [BT00]. Because MBT enables the automatic generation and execution of test-cases and the evaluation of the outcome of tests, it makes it possible to test more thoroughly and possibly cheaper than by manual testing.

Before we start talking about model-based testing, we want to make more precise what kind of software testing game we are in. The work in this thesis is in the tradition of LTS based testing. To be more precise, our research is in the tradition of conformance testing [BAL$^+$90, Tre94, ISO96, Tre99] and uses the **ioco** test theory of Tretmans [Tre96b, Tre08]. We favor LTS-based testing because we find it poses less restrictions on the model we use. For example FSM-based testing requires deterministic systems, synchronous communication of input and output actions and it needs an estimate of the number of states in the implementation [LY96]. We are aware of efforts to lift or lessen these restrictions, but to our knowledge, so far they come with the price of other or extra restrictions [Pet00]. Relevant for this thesis, FSMs require extra effort to support parallel composition (due to the nature of the coupling of input and output actions on a transition). LTSs support parallel composition in a simple and elegant way (see Section 2.3.2). This is not the case for **ioco**, as we will find out in the next chapter.

The aim of this chapter is to introduce the concepts and ideas used in (formal) model-based testing and especially the concepts used in the **ioco** theory. We put the **ioco** theory into perspective with respect to other theories, by treating some test theories that influenced the **ioco** theory. The purpose of this chapter is to give all the background necessary to read this thesis. It is organized around the concepts of model-based testing in the following way: we introduce several classes of label transition systems in Section 2.3, **ioco** and several other *notions of correctness* in Section 2.4 and *test-cases* in Section 2.5. A good portion of the material in this chapter is reused from the chapter "I/O Automata Based Testing" written together with F. Peureux [vdBP04] for the book *Model-based testing of reactive systems* [BJK$^+$05]. We start with a framework by Tretmans to introduce formal methods in conformance testing [Tre02].

2.2 Framework for conformance testing

In this section we present a framework, depicted graphically in Figure 2.1 on the facing page, for conformance testing. Our aim is to introduce and for-

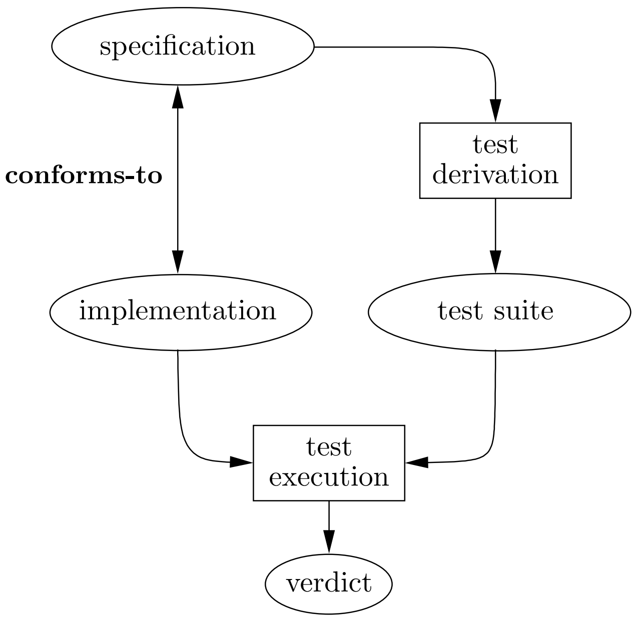


Figure 2.1: Formal Conformance Testing Framework

malize the MBT concepts that we will use in this thesis. In the figure we see the objects: specification, implementation, test suite, verdict, and the activities: test derivation and test execution. We also see a *conformance relation* between the specification and the implementation. This relation expresses under what conditions the implementation conforms to, i.e., is correct with respect to the specification. In order to find out if an implementation conforms to a specification we perform experiments on the implementation. In the world of hardware and software testing we call these experiments *tests*; we call the specification of a test a *test-case*. A collection of test-cases is called a *test suite*. With the aid of a test derivation algorithm we derive test-cases from the specification. The execution of a test-case leads to a verdict whether the implementation conforms to the specification. We identify two verdicts: **pass** and **fail**.

Conformance is a notion of correctness between a specification and an implementation. In our formal framework we use formal specifications, i.e., mathematical objects. We refer to a formal specification by SPEC and we denote the universe of formal specifications by *SPECS*. Implementations are real world entities, in general hardware/software combinations. They are the systems that we are going to test and we refer to them as IUT (Implementation Under Test). We denote the universe of IUTs as *IMPS* (Implementations). Conformance could be introduced as a relation **conforms-to** $\subseteq$ *IMPS* $\times$ *SPECS*, with ‘IUT **conforms-to** SPEC’ expressing that the IUT is a correct implementation of the specification SPEC. However is is impossible to give a formal definition of **conforms-to** as IUTs are not formal objects. In order to reason formally about implementations we make the assumption that any real implementation IUT $\in$ *IMPS* can be modeled by a formal object $i_{\text{IUT}} \in \mathit{MODS}$, where *MODS* is the universe of implementation models. This assumption is known as the *test hypothesis* [Ber91]. It is a necessary theoretical step to connect the formal and physical world. For practical testing we do not have to identify this model, i.e., the element in *MODS*, concretely for a given implementation to test it.

The test hypothesis makes it possible to express conformance as a formal relation between models of implementations and specifications. Such a relation is called an implementation relation: $\mathbf{imp} \subseteq \mathit{MODS} \times \mathit{SPECS}$ [BAL^{+}90, ISO96]. We say that implementation IUT $\in$ *IMPS* is correct with respect to specification SPEC $\in$ *SPECS* (IUT **conforms-to** SPEC), if and only if the model of the implementation, i_{IUT} is **imp**-related to SPEC: i_{IUT} **imp** SPEC; formally:

$$\text{IUT } \mathbf{conforms\text{-}to} \text{ SPEC} \Leftrightarrow i_{\text{IUT}} \ \mathbf{imp} \text{ SPEC}$$

Testing is the execution of test-cases on the implementation. We denote the universe of test-cases by *TESTS*. We denote the execution of a test-case $t \in \mathit{TESTS}$ on an implementation IUT $\in$ *IMPS* by $\mathit{EXEC}(t, \text{IUT})$. During test execution we stimulate the IUT with actions, for example by pressing a button on a keyboard, and as a result we may observe responses from the IUT. We denote the domain of all observations by *OBS*. Test execution $\mathit{EXEC}(t, \text{IUT})$ results in a subset of *OBS*. We use 2^{OBS} to denote the set of subsets of *OBS*.

$\mathit{EXEC}(t, \text{IUT})$ takes place in the physical world. In order to reason formally about test execution we model this process in our formal domain. We do this by introducing a formal observation function $\mathit{obs} : \mathit{TESTS} \times \mathit{MODS} \to 2^{\mathit{OBS}}$. So $\mathit{obs}(t, i_{\text{IUT}})$ formally models the real test execution $\mathit{EXEC}(t, \text{IUT})$. Now we can state more precisely what we mean with the test hypothesis: For all IUT in *IMPS* there exists a model i_{IUT} in *MODS* such that for $t \in \mathit{TESTS}$ $\mathit{EXEC}(t, \text{IUT})$ equals $\mathit{obs}(t, i_{\text{IUT}})$

In words this states: for all physical implementations, it is assumed that there is a model of this implementation, such that if we execute all tests in *TESTS*, then we cannot distinguish the implementation from the model. This notion is analogous to the ideas underlying testing equivalences [DNH84, DN87].

In order to explain the testing concepts in a straightforward and concise manner we swept some details under the rug. As the observant reader has probably noticed, we use the test-cases in *TESTS*, the observations in *OBS* and the specifications in *SPECS* in the physical and the formal world. A more correct approach would be to distinguish the formal objects from the physical objects, like we did for the implementation. We could do this, but we find that it makes our explanation unnecessarily complex.

The purpose of test execution is to give a verdict about the correctness of the IUT. To reason formally about verdicts we introduce a verdict function $\mathit{verd} : \mathit{TESTS} \times 2^{\mathit{OBS}} \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$. This verdict function expresses for a certain test t which observations are correct. It is common to talk in terms of verdicts on test-cases instead of observations. We say that an IUT passes a test-case t if the verdict of the test execution is **pass**. We define this formally as follows:

$$\textsc{iut}\ \mathbf{passes}\ t =_{\mathrm{def}} \mathit{verd}(t, \mathit{EXEC}(t, \textsc{iut})) = \mathbf{pass}$$

Likewise we write IUT **fails** t to denote IUT $\mathbf{pas\not{s}es}$ t (we take the liberty of denoting negation by slashing).

Conformance testing In conformance testing we use an implementation relation as a formal notion of correctness to judge if an implementation conforms to its specification. With this implementation relation we can derive test-cases with verdicts from the specification. We execute the test-cases against the IUT in order to check if the IUT conforms to the specification. To put this approach into practice we link the notions of conformance and of test execution (expressed by EXEC) in such a way that test execution gives us an indication of conformance. Ideally, given specification $\textsc{spec} \in \mathit{SPECS}$, we would like to have a test suite $T \subseteq \mathit{TESTS}$ such that the following holds:

$$\textsc{iut}\ \mathbf{conforms\text{-}to}\ \textsc{spec} \Leftrightarrow \textsc{iut}\ \mathbf{passes}\ T$$

A test suite with this property is called *complete*. It can distinguish exactly between all conforming and non-conforming implementations. In practice a complete test suite is very big if not infinite. The cause is primarily in the right-to-left implication of the formula, which we call *exhaustiveness*. It states that we have a conforming IUT if it passes the test suite. In other words the test suite needs to take all possible errors into account, these may be very many (remember the calculator example from the introduction of this thesis). The implication from left to right is called *soundness*. Soundness is an important requirement. It states that if a test-case reports a failure, then we really have a non-conforming implementation (i.e., there is an error in the implementation).

An important activity in conformance testing is *test-case derivation*. Formally, test derivation can be seen as a function $\mathit{der} : \mathit{SPECS} \rightarrow 2^{\mathit{TESTS}}$. Such a function should produce at least sound test suites and if possible exhaustive test suites. Exhaustiveness in practice often requires an unlimited amount of time and resources. Nonetheless, we do find this property important because it does not a priori leave out important test-cases. From a theoretical perspective, if we let run an exhaustive test-generation procedure ad infinitum, we might call it limit-complete. We find this better than a procedure that is incomplete even in the limit.

From a practical point of view it is mostly impossible to say if an implementation conforms to a specification, because we need a complete test suite. In the case that such a test suite is (practically) infinite we cannot answer the conformance question. Hence the famous quote by Dijkstra (already in 1969) that “testing can be used to show the presence of bugs, but never to show their absence” [Dij69]. So what is the practical use of conformance testing? The best we can do in practice is to have a sound test suite with a good coverage of the functionality. Practice shows that when a system passes a test suite with a good coverage, this is an indication that there are no obvious mistakes in the system. A test suite with a good coverage is in most cases still a very big test suite and humans are notoriously bad in creating good test suites. We find that the conformance testing theory gives a good basis to construct these kind of test suites.

Conclusion We have treated the parts of the formal testing framework individually. We briefly want to recapitulate how the parts of the framework work together.

- We have a formal specification SPEC that describes the desired behavior of the software system.
- We have an implementation IUT. To make the test theory work, we assume that it can be adequately represented by a formal model.
- We want to know if the IUT is a conforming, in other words correct, implementation of SPEC. In order to find out if the IUT is correct we execute test-cases against the IUT.
- We use the information in SPEC in order to generate test-cases. Important properties of test suites are soundness, exhaustiveness and completeness. Only a complete test suite can distinguish between all conforming and non-conforming implementations.
- We execute the generated test suite against the IUT.
- Based on the observations of the test execution we give the verdict **pass** or **fail**.

In Table 2.1 we give an overview of the concepts.

Physical ingredients:	
Black box implementation	$\text{IUT} \in \mathit{IMPS}$
Execution of a test	$\mathit{EXEC}(t, \text{IUT})$
Formal ingredients:	
Specification:	$\text{SPEC} \in \mathit{SPECS}$
Implementation model	$i_{\text{IUT}} \in \mathit{MODS}$
Implementation relation:	$\mathbf{imp} \subseteq \mathit{MODS} \times \mathit{SPECS}$
Test-case:	$t \in \mathit{TESTS}$
Test suite:	$T \in 2^{\mathit{TESTS}}$
Observations:	OBS
Formal test execution:	$\mathit{obs} : \mathit{TESTS} \times \mathit{MODS} \to 2^{\mathit{OBS}}$
Verdict:	$\mathbf{pass}, \mathbf{fail}$
Verdict function:	$\mathit{verd} : \mathit{TESTS} \times 2^{\mathit{OBS}} \to \{\mathbf{pass}, \mathbf{fail}\}$
Test derivation:	$\mathit{der} : \text{SPEC} \to 2^{\mathit{TESTS}}$
Assumptions:	
Test hypothesis:	IUT can be modeled by $i_{\text{IUT}} \in \mathit{MODS}$
	$\mathit{obs}(t, i_{\text{IUT}})$ models $\mathit{EXEC}(t, \text{IUT})$
Proof obligation:	
Soundness:	IUT **fails** $T \Rightarrow \neg(\text{IUT}$ **conforms-to** SPEC$)$
Exhaustiveness:	IUT **passes** $T \Rightarrow$ IUT **conforms-to** SPEC

Table 2.1: Formal Model-Based Testing ingredients

2.3 Labeled transition system models

In this section we present formalisms for specifications and implementations. Our research is in the tradition of testing with Labeled Transition Systems (LTS), therefore all our formalisms are based on LTSs.

We introduce the general LTS model and a variant of the LTS model, the IOLTS, that distinguishes inputs and outputs together with some standard notation and definitions in Section 2.3.1. To create systems directly with the LTS model can be a laborious exercise. In Section 2.3.2 we introduce a process language that makes it easier to create and notate large systems. In Section 2.3.3 we present two types of transition systems to model implementation behavior.

2.3.1 Labeled transition systems

A labeled transition system (LTS) is defined in terms of states and labeled transitions between states, where the labels indicate what happens during the transition. Labels are taken from a countable global set $\mathbf{L}$; these are so called *observable* actions. We use a special label $\tau \notin \mathbf{L}$ to denote an internal or hidden action. For arbitrary $L \subseteq \mathbf{L}$, we use L_τ as a shorthand for $L \cup \{\tau\}$.

Definition 2.3.1 [Labeled Transition System] A labeled transition system is a 4-tuple $\langle Q, L, T, \mathsf{start}\rangle$, where

- Q is a countable, non-empty set of states;
- $L \subseteq \mathbf{L}$ is a countable set of labels;
- $T \subseteq Q \times L_\tau \times Q$ is the transition relation;
- $\mathsf{start} \in Q$ is the start state.

In this thesis we use LTSs that make a distinction between input and output actions. We call such a system an Input Output Labeled Transition System (IOLTS). When the (input-output) context is clear we may use the term LTS for an IOLTS.

Definition 2.3.2 [Input Output Labeled Transition System] An IOLTS is an LTS where the label set L is partitioned into an input label set I and an output label set U. Formally it is a 5-tuple $\langle Q, I, U, T, \mathsf{start}\rangle$ where Q is a countable, non-empty set of *states*; $I \subseteq \mathbf{L}$ is the countable set of *input labels*; $U \subseteq \mathbf{L}$ is the countable set of *output labels*, which is disjoint from I; $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ is the *transition relation*; $\mathsf{start} \in Q$ is the *initial state.*

We use L as shorthand for the entire label set ($L = I \cup U$) and we use Q_s, I_s etc. to refer to the components of an (IO)LTS s. We commonly write $q \xrightarrow{\mu} q'$ for $(q, \mu, q') \in T$. We use a question mark before a label to denote that the label is an input action and an exclamation mark to denote that the label is an output action. We denote the class of all labeled transition systems over L by $\mathbf{LTS}(L)$, likewise we denote the class of all IOLTSs over I and U by $\mathbf{IOLTS}(I, U)$. We represent a labeled transition system in the standard way, by a directed, edge-labeled graph where nodes represent states and edges represent transitions (see Example 2.3.6 for an example).

A state from which no internal action is possible is called *stable*. A stable state from which no output action is possible is called *quiescent*. We use the symbol δ ($\notin \mathbf{L}_\tau$) to represent quiescence: that is, $q \xrightarrow{\delta} q$ stands for the absence of any transition $q \xrightarrow{x} q'$ with $x \in U_\tau$. For an arbitrary $L \subseteq \mathbf{L}$, we use L_δ as a shorthand for $L \cup \{\delta\}$. Likewise we use L_τ as a shorthand for $L \cup \{\tau\}$. The notation $\delta(q)$ denotes that the state q is quiescent.

An LTS is called *strongly convergent* if it does not have infinite compositions of internal actions; in other words, if it does not have any infinite τ-labeled paths. For technical reasons we restrict the fragment we use of $\mathbf{IOLTS}(I, U)$ to strongly convergent transition systems (this is a restriction of the **ioco** theory).

A *trace* is a finite sequence of observable actions. The set of all traces over L ($\subseteq \mathbf{L}$) is denoted by L^*. When δ and/or τ are part of the label set we show this explicitly by using subscripts; for example the set of traces $L^*_{\delta\tau} = (L \cup \{\delta, \tau\})^*$ for some $L \subseteq \mathbf{L}$. Traces are ranged over by σ, with ϵ denoting the empty sequence. We will use Σ to denote a set of traces. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. Concatenation is extended in the standard way to sets of traces, denoted by $\Sigma_1 \cdot \Sigma_2$ (with Σ_1, Σ_2 sets of traces). We use the standard notation with single and double arrows for traces: $q \xrightarrow{\lambda_1 \cdots \lambda_n} q'$ denotes $q \xrightarrow{\lambda_1} q_1 \cdots q_{n-1} \xrightarrow{\lambda_n} q'$, $q \overset{\epsilon}{\Longrightarrow} q'$ denotes $q \xrightarrow{\tau \cdots \tau} q'$ and $q \overset{\lambda_1 \cdots \lambda_n}{\Longrightarrow} q$ denotes $q \overset{\epsilon}{\Longrightarrow} \xrightarrow{\lambda_1} \overset{\epsilon}{\Longrightarrow} \cdots \xrightarrow{\lambda_n} \overset{\epsilon}{\Longrightarrow} q'$. We write $q \xrightarrow{\mu}$ as a shorthand for $\exists q' \in Q : q \xrightarrow{\mu} q'$. We lift this notation in a straightforward manner to traces and the double arrow notation. An *execution fragment* of a transition system s is a sequence of alternate states and actions $\alpha = q_0 a_1 q_1 a_2 q_2 \ldots$, starting with a state and if the execution fragment is finite ending with a state, where each $(q_i, a_{i+1}, q_{i+1}) \in T_s$ for $i \geq 0$. An *execution* is a fragment that starts in the start state. When we refer to the trace of an execution (fragment), we mean the execution (fragment) without the states (the result is a trace). When it does not lead to confusion we will not always distinguish between a labeled transition system and its initial state. We will identify the system $s = \langle Q, I, U, T, \mathsf{start} \rangle$ with its initial state start, and we write, for example, $s \overset{\sigma}{\Longrightarrow} q_1$ instead of $\mathsf{start} \overset{\sigma}{\Longrightarrow} q_1$. In Definition 2.3.3 we repeat some standard definitions for LTSs, likewise in Definition 2.3.4 for IOLTSs.

Definition 2.3.3 Let $s = \langle Q, L, T, \mathsf{start} \rangle \in \mathbf{LTS}(L)$, $\sigma \in L^*_\delta$, $q \in Q$ and $Q' \subseteq Q$.

- $\mathit{init}(q) =_{\text{def}} \{\mu \in L_\tau \mid q \xrightarrow{\mu}\}$
- q **after** $\sigma =_{\text{def}} \{q' \mid q \overset{\sigma}{\Longrightarrow} q'\}$
- Q' **after** $\sigma =_{\text{def}} \bigcup_{q' \in Q'} (q'$ **after** $\sigma)$
- $\mathit{traces}(s) =_{\text{def}} \{\sigma' \in L^* \mid s \overset{\sigma'}{\Longrightarrow}\}$
- s is *deterministic* if forall $\sigma' \in L^*_\delta$, s **after** σ' has at most one element.
- s has finite behavior if there is a natural number n, such that the length of all traces in $\mathit{traces}(s)$ is smaller than n.

init(q) is the set of initial actions of a state q. q **after** σ is the set of states reachable from state q by performing the trace σ. *traces*(s) is the set of traces that an LTS s can perform. A transition system is deterministic if no more than one state is reachable with a trace. This means that if there is a trace σ that leads to two or more states, starting in a state q, the system is not deterministic. These definitions are extended in a straightforward way to IOLTSs.

Definition 2.3.4 Let $s = \langle Q, I, U, T, \mathsf{start}\rangle \in \mathbf{IOLTS}(L)$, $\sigma \in L_\delta^*$, $q \in Q$ and $Q' \subseteq Q$.

- $\mathit{qtraces}(s) =_{\mathrm{def}} \{\sigma' \in L^* \mid \exists q' \in Q : s \overset{\sigma'}{\Longrightarrow} q' \wedge \delta(q')\}$
- $\mathit{Straces}(s) =_{\mathrm{def}} \{\sigma' \in L_\delta^* \mid s \overset{\sigma'}{\Longrightarrow}\}$
- $\mathit{out}(q) =_{\mathrm{def}} \{x \in U_\delta \mid q \overset{x}{\Longrightarrow}\}$
- $\mathit{out}(Q') =_{\mathrm{def}} \bigcup_{q \in Q'} \mathit{out}(q)$

qtraces are traces that end in a quiescent state. *Straces*(s) are the *suspension* traces that an LTS s can perform. Suspension traces are traces that may include the action δ. *out*(q) is the set of outputs, including quiescence, possible in state q, or after ϵ. The out definition is extended in a straightforward manner to sets of states.

Projection is a way to remove unwanted labels from a trace. In the definition below, labels in Σ are untouched and labels not in Σ are replaced by ϵ.

Definition 2.3.5 [Projection] Let $\lambda \in L_{\tau\delta}, \Sigma \subseteq L_{\tau\delta}$.

$$\lambda\lceil\Sigma = \begin{cases} \epsilon & \text{if } \lambda \notin \Sigma \\ \lambda & \text{otherwise} \end{cases}$$

We extend the definition of projection to traces in the following way. Let $\sigma = \lambda_1 \cdots \lambda_n$ for some $n \geq 1$ with $\forall 1 \leq i \leq n : \lambda_i \in L_{\tau\delta}$. $(\lambda_1 \cdots \lambda_n)\lceil\Sigma = (\lambda_1\lceil\Sigma \cdots \lambda_n\lceil\Sigma)$

Example 2.3.6 In Figure 2.2 on the next page we show an example of an IOLTS s. It models the behavior of a coffee machine. After we press $\mathsf{button1}$, an internal step is executed (for example heating the water, or initializing the machine) and we get a cup of coffee. Likewise, when we press $\mathsf{button2}$ we get a cup of tea. Formally, $s = \langle Q, I, U, T, \mathsf{start}\rangle$ with:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $I = \{\mathsf{button1}, \mathsf{button2}\}$
- $U = \{\mathsf{coffee}, \mathsf{tea}\}$

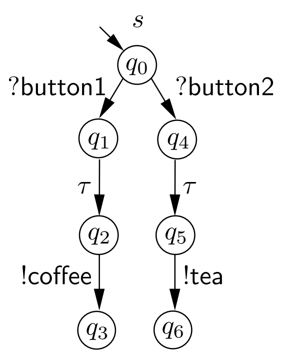


Figure 2.2: Example of an IOLTS

- $T = \{(q_0, \mathsf{button1}, q_1), (q_1, \tau, q_2), (q_2, \mathsf{coffee}, q_3), (q_0, \mathsf{button2}, q_4), (q_4, \tau, q_5), (q_5, \mathsf{tea}, q_6)\}$
- $\mathsf{start} = q_0$

The state q_0 is stable and quiescent ($\delta(q_0)$) as there are no outgoing τ and output transitions. We write $s \xrightarrow{\mathsf{button1}\cdot\tau\cdot\mathsf{coffee}} q_3$ and when we want to abstract from τ actions we may write $s \xRightarrow{\mathsf{button1}\cdot\mathsf{coffee}} q_3$. The initial actions of s are: $init(s) = \{\mathsf{button1}, \mathsf{button2}\}$. For the set of traces and suspension traces we have $traces(s) = \{\epsilon, \mathsf{button1}, \mathsf{button2}, \mathsf{button1}\cdot\mathsf{coffee}, \mathsf{button2}\cdot\mathsf{tea}\}$ and $Straces(s) = traces(s) \cup \delta^* \cup \delta^*\cdot\mathsf{button1}\cdot\mathsf{coffee}\cdot\delta^* \cup \delta^*\cdot\mathsf{button2}\cdot\mathsf{tea}\cdot\delta^*$. start **after** $\mathsf{button1} = \{q_1, q_2\}$. When we combine the definitions of *out* and **after** we get $out(s$ **after** $\mathsf{button1}) = \{\mathsf{coffee}\}$. □

Note that in Figure 2.2 we show the state names in the state. When we abstract from state names, we represent states by black dots.

2.3.2 Representing labeled transition systems

Except for relatively small systems, a representation by graphs or trees, like with LTS models, is laborious. "Real world" systems easily have thousands of states which makes drawing them practically impossible. A standard way to represent an LTS is a process (algebraic) language. In this thesis we sometimes use processes to model system behavior. In this section we introduce the syntax and semantics of our process language, which is a variant of the language LOTOS [BB87, ISO89]. A more detailed treatment of process algebras can be found in [Hoa85, Mil89, BB87, ISO89]. It is not our intention to use this language in a mathematical way and to prove properties of it. Our main purpose is to have a convenient and concise notation for LTSs.

Before we treat the syntax and semantics of our language, we start with its parameters. We distinguish actions and process names. Similar to LTSs

Operator	Transition rules
$a; B$	$\dfrac{}{a; B \xrightarrow{a} B}$
$\Sigma\mathcal{B}$	$\dfrac{\exists B \in \mathcal{B} : B \xrightarrow{\mu} B'}{\Sigma\mathcal{B} \xrightarrow{\mu} B'}$
hide V **in** B	$\dfrac{B \xrightarrow{\mu} B', \mu \in V}{\textbf{hide}\ V\ \textbf{in}\ B \xrightarrow{\tau} \textbf{hide}\ V\ \textbf{in}\ B'}$ $\dfrac{B \xrightarrow{\mu} B', \mu \notin V}{\textbf{hide}\ V\ \textbf{in}\ B \xrightarrow{\mu} \textbf{hide}\ V\ \textbf{in}\ B'}$
$B_1 \parallel_G B_2$	$\dfrac{B_1 \xrightarrow{\mu} B_1', \mu \notin G}{B_1 \parallel_G B_2 \xrightarrow{\mu} B_1' \parallel_G B_2}$ $\dfrac{B_2 \xrightarrow{\mu} B_2', \mu \notin G}{B_1 \parallel_G B_2 \xrightarrow{\mu} B_1 \parallel_G B_2'}$ $\dfrac{B_1 \xrightarrow{\mu} B_1', B_2 \xrightarrow{\mu} B_2', \mu \in G}{B_1 \parallel_G B_2 \xrightarrow{\mu} B_1' \parallel_G B_2'}$
$P := B$	$\dfrac{B \xrightarrow{\mu} B'}{P \xrightarrow{\mu} B'}$
stop	no rules

Table 2.2: Transition rules for the process language operators

we assume a fixed, countable universe of actions and distinguish the internal action τ. A process name allows us to refer to processes by name.

We define a set of behavior expressions $\mathcal{B}(L)$ over L. We use L_B to refer to the label-set of behavior expression B. We assume the label-set to be fixed when the behavior expression is created, i.e., the label-set does not change during the lifetime of a behavior expression. We use the following syntax for a behavior expression B, let $\mathcal{B}$ be a set of behavior expressions, where $a \in L \cup \{\tau\}$:

$$B =_{\text{def}} a; B \mid \Sigma\mathcal{B} \mid \textbf{hide}\ V\ \textbf{in}\ B \mid B_1 \parallel_G B_2 \mid P := B \mid \textbf{stop}$$

In Table 2.2 we show the operational semantics for our process language in the Structural Operational Semantics (SOS) style introduced by Plotkin [Plo81]. It describes the transition rules to go from one state to another (and hence to build an LTS). At the end of this section we give an example how to come from a process definition to a transition system.

Action prefix Action prefix is denoted by ';'. The expression $a; B$ with $a \in L_\tau$ describes the behavior that can perform the action a and then behaves as B. The SOS rule for action prefix prescribes that the process $a; B$ can make a transition to process B. In other words we can interpret $a; B$ as a transition system that can make a transition labeled with a and continue as process B.

Choice $\Sigma\mathcal{B}$, where $\mathcal{B}$ is a countable set of behavior expressions, denotes a *choice* of behavior. It behaves as any of the processes in $\mathcal{B}$. We use the operator '+' to denote choice between two behavior expressions. The SOS rule for choice says that the system $\Sigma\mathcal{B}$ can make a transition with action a to B' if there is a process in $B \in \mathcal{B}$ such that $B \xrightarrow{a} B'$.

Hiding For a set of actions V and a process B, the expression **hide** V **in** B means that the actions in label set V are hidden in process B. This means that if B can do a transition with a label $x \in V$, the action becomes invisible. Actions not in V remain visible.

Parallel composition $B_1 \parallel_G B_2$, where $G \subseteq L$, denotes the parallel composition of B_1 and B_2. In this parallel composition all actions in G must *synchronize*, i.e., they must occur in both processes at the same time. All actions not in G (including τ) are *interleaved*, i.e., they can occur independently in both processes.

We use $\parallel$ as an abbreviation for synchronization on the actions in the intersection of the label sets of the processes involved. This means that $B_1 \parallel B_2 = B_1 \parallel_G B_2$ with $G = L_{B_1} \cap L_{B_2}$. In this thesis we primarily use parallel composition with systems with input and output actions. In that case we synchronize inputs with outputs: $G = (I_{B_1} \cap U_{B_2}) \cup (U_{B_1} \cap I_{B_2})$, where we assume that $I_{B_1} \cap I_{B_2} = U_{B_1} \cap U_{B_2} = \emptyset$. The synchronized result is an output action. The signature of the resulting system $B_1 \parallel_G B_2$ has $I = (I_{B_1} \cup I_{B_2}) \backslash G$ and $U = U_{B_1} \cup U_{B_2}$.

Inaction A process that does nothing is denoted by **stop**. It amounts to deadlock. Sometimes the notation $\Sigma\emptyset$ is used in the literature to define inaction.

Process instantiation *Process definition* assigns a process name to a behavior expression. Process definition P behaves as B, where P is defined by $P := B$. Process definition makes it easier to refer to a more complex behavior expression, including recursive and repetitive behavior.

Example 2.3.7 Suppose we want to model the coffee machine in Figure 2.2 on page 21 in the presented process language. We can do this as follows. We identify two parts, one for producing coffee and the other for producing tea. The coffee making process is defined by pressing $\mathsf{button1}$, followed by some internal action, after which coffee is produced. We can model this by $C := \mathsf{button1}; \tau; \mathsf{coffee}; \mathbf{stop}$. We show the transition system for C on the left-hand side in Figure 2.3 on the following page. Next to the transitions we show the actions and next to the states we show the behavior expression (in a smaller font). We can read the first transition as follows (application of the action prefix rule in Table 2.2): $\mathsf{button1}; \tau; \mathsf{coffee}; \mathbf{stop} \xrightarrow{\mathsf{button1}} \tau; \mathsf{coffee}; \mathbf{stop}$. Likewise we can model the tea making process by $T := \mathsf{button2}; \tau; \mathsf{tea}; \mathbf{stop}$.

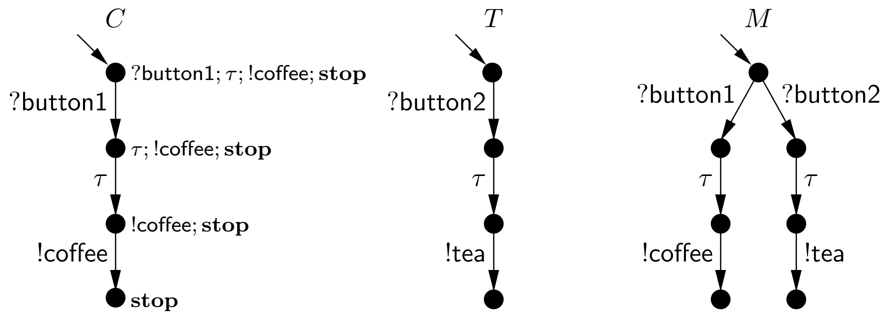


Figure 2.3: Process language example

We show the corresponding transition system for this process in the middle of the figure (we only show the actions for T, not the states). We combine the coffee and tea making processes into one coffee machine M by using the choice operator: $M := C + T$. This results in the transition system on the right-hand side. □

2.3.3 Input-enabled transition systems

An (IO)LTS is in general used to model specification behavior. There are limitations to using an (IO)LTS as a model for implementation behavior: an (IO)LTS can *block* inputs from the environment. Blocking of an input action occurs in an (IO)LTS when we enter a state that has no outgoing transitions for this action, as a result we cannot process the input action. For many implementations blocking is unwanted or unrealistic behavior, for example we can always press a button on our TV remote control and we can always send a command when using a communication protocol. To remedy this problem, *input-enabled* models were invented. These models are in principle transition systems with a special property, that inputs from the environment cannot be blocked. In this section we treat two input enabled models. For this thesis, the *Input Output Transition System* by Tretmans [Tre96b] is the important model. It was influenced by the *Input Output Automata* (IOA), introduced by Lynch and Tuttle [LT89].

The general notion underlying these models is the distinction between actions that are locally controlled and actions that are not locally controlled. Output and internal actions of a transition system are locally controlled. This means that these actions are performed autonomously, i.e., independent of the environment. Inputs on the other hand, are not locally controlled; they are under control of the environment. This means that the system can never block an input action; this property is called input-enabledness or input-completeness. We distinguish two variants of input-enabledness: *strong* and *weak* input-enabledness. Strong input-enabledness requires that all input actions are enabled in all states. Weak input-enabledness requires that all input actions can be performed from all *stable* states.

Input-output automata The input-output automaton was introduced by Lynch and Tuttle in 1987 [LT87], [LT89]. An automaton's actions are classified as either 'input', 'output' or 'internal'. Communication of an IOA with its environment is performed by synchronization of output actions of the environment with input actions of the IOA and vice versa. Because locally controlled actions are performed autonomously, it is required that input actions can never be blocked. Therefore an IOA is input enabled (it can process all inputs in every state). In order to compare the models we use a uniform presentation (based on the IOLTS notation). As a result our notation for the IOA model differs from the notation found in the literature.

Definition 2.3.8 [I/O automaton] An I/O automaton $s = \langle Q, I, U, H, T, \mathsf{start}, P\rangle$, where:

- Q is a set of states,
- $\mathsf{start} \in Q$ is the start state,
- $I \subseteq \mathbf{L}$ is a set of input actions,
- $U \subseteq \mathbf{L}$ is a set of output actions,
- $H \subseteq \mathbf{L}$ is a set of internal (or **H**idden) actions. $H \cap I = H \cap U = \emptyset$. A transition $q_1 \xrightarrow{a} q_2$ with $q_1, q_2 \in Q$ and $a \in H$ manifests as $q_1 \xrightarrow{\tau} q_2$ in the IOLTS.
- $T \subseteq Q \times (I \cup U \cup H) \times Q$ is the transition relation.
- P is an equivalence relation that partitions the set of locally controlled actions $U \cup H$ into at most a countable number of equivalence classes.
- s is strongly input-enabled. Formally: $\forall q \in Q, a \in I : q \xrightarrow{a}$

An IOA is an IOLTS with the exception that it has a set of internal actions H, an equivalence relation P, and is input-enabled. The set of internal actions is *not observable* by the environment, but works otherwise like the other actions (an IOLTS abstracts all internal actions to τ actions). The equivalence relation P is only used in the definition of fair computation that is used in the fair pre-order (Definition 2.4.3); we discuss it after Example 2.3.9.

The extra label set H may create some notational confusion. In this thesis we are only interested in observable actions, therefore we use L for the set of so called *external* actions: $I \cup U$. *traces*(s) of an IOA s denotes the set of external traces; traces that do not have internal actions.

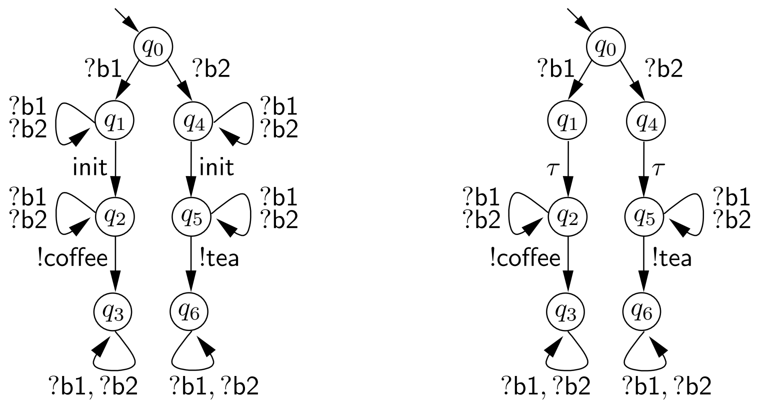


Figure 2.4: Examples of an IOA and IOTS

Example 2.3.9 In Figure 2.4 we show an IOA (left-hand side) and an IOTS model (right-hand side). All transition systems model a coffee machine. We focus on the IOA and discuss the IOTS later on. We can push two buttons: button1 (abbreviated to b1 in the figure) and button2 (abbreviated to b2 in the figure). After pushing button1 the machine initializes (init) and outputs coffee, and after pushing button2 the machine initializes and outputs tea. button1 and button2 are input actions, coffee and tea are output actions and init is an internal action. The self-loops with b1 and b2 in states q_1 till q_6 show that the automaton is input enabled in every state. q_0 does not need these self-loops, since all input actions, button1 and button2, are enabled in this state. □

A possible problem with the input-output automata model is that an automaton cannot give an output action, because it has to handle a never ending stream of input actions. Since it is input-enabled, it is always able to synchronize on an input from the environment. Lynch and Tuttle therefore introduce a notion of fairness for IOA. In short this means that a locally controlled action (i.e., an output or internal action) cannot be blocked by input actions forever. The partitioning P of the locally controlled actions is used in the operationalization of fairness. Note that the problem of fairness exists for all models that implement the notion of *input enabledness*.

An execution α of an IOA s is *fair* if either α ends in a *quiescent* state or α is infinite and for each class $c \in P(s)$ either actions from c occur infinitely often in α or states from which no action from c is enabled appear infinitely often in α. A fair trace of an IOA s is the trace (with only external actions) of a fair execution of s. To put it in words, a trace is fair 1) if it is finite and ends in a quiescent state, 2) if it is infinite and there is no state that it encounters infinitely often for which a locally controlled action is blocked. The set of fair traces of an IOA s is denoted by $\mathit{Ftraces}(s)$. Contrary to all other trace definitions in this thesis, the set of *Ftraces* may contain traces of infinite length. We illustrate fairness in the following example.

Example 2.3.10 Let us look again at the IOA in Figure 2.4 on the facing page (left-hand side). Let $I = \{\mathsf{button1}, \mathsf{button2}\}$, $U = \{\mathsf{coffee}, \mathsf{tea}\}$, $H = \{\mathsf{init}\}$, $P = \{\{\mathsf{init}, \mathsf{coffee}, \mathsf{tea}\}\}$. P is the trivial partitioning of locally controlled actions. The trace button1 is not a fair trace, as it does not end in a quiescent state (it ends in q_1 or q_2). The trace button1·init·coffee is a fair trace because it ends in q_3, a quiescent state. All traces button1·init·button1* (button1 followed by init followed by zero or more times button1) are not fair. The finite traces in the set are not fair because they end in q_2, which is not a quiescent state. The infinite traces in the set are not fair because they encounter the state q_2 infinitely often, but the locally controlled actions in q_2 (coffee) do not occur in the trace. □

Input-output transition system The input-output transition system was introduced by Tretmans [Tre96b]. It is basically a simplified version of the IOA: it does not have an equivalence relation and it models internal actions with the τ label. A subtle but important difference is that an IOTS is weakly input enabled: $\forall a \in I, q \in Q : q \overset{a}{\Longrightarrow}$. We denote the class of input-output transition systems over I and U by $\mathbf{IOTS}(I, U)$.

Definition 2.3.11 [Input-Output Transition System]
An *Input-Output Transition System* $s = \langle Q, I, U, T, \mathsf{start}\rangle$ is a weakly input-enabled IOLTS. Formally this means: $\forall q \in Q, a \in I : q \overset{a}{\Longrightarrow}$.

Example 2.3.12 In Figure 2.4 on the preceding page, the transition system on the right-hand side is an IOTS. We see that the internal action init is replaced by τ. Notice furthermore that the (non-stable) states q_1 and q_4 do not have the self-loops with button1 and button2. This is allowed because an IOTS is weakly input enabled. With an internal action we can go from q_1 to the input enabled state q_2 (the same holds for q_4 and q_5). □

2.4 Input output implementation relations

In this section we introduce the **ioco** theory and the **ioco** implementation relation. The **ioco** theory was influenced by several other theories, amongst them implementation relations defined on IOA and LTS ([DNH84, Abr87, Bri87, Phi87, LT89, Lan90, Seg93, Pha94]). Two important characteristics of the **ioco** theory are that it is LTS-based and that it uses input-enabled models. This can be traced back to De Nicola and Hennessy ([DNH84])and Lynch and Tuttle ([LT89]), respectively. Segala compared the IOA model and the theory of testing of De Nicola and Hennessy and defined the so called MAY and MUST pre-orders (that we treat in Section 2.4.1) directly on

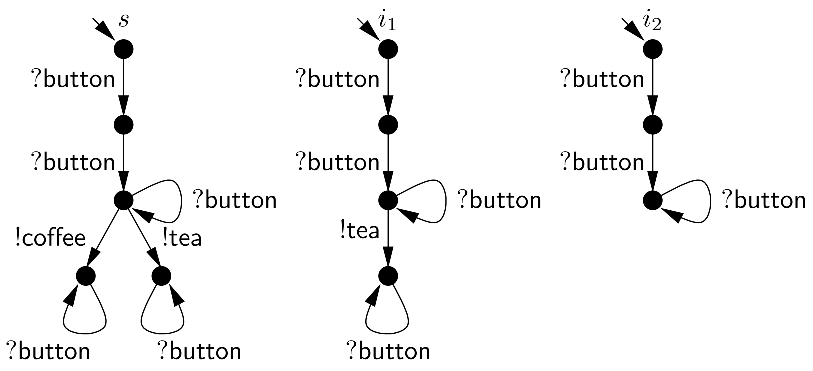


Figure 2.5: Example of the trace inclusion pre-order

IOA ([Seg97]). In order to put the **ioco** theory into perspective with respect to other theories, and to identify its unique characteristics, we give a quick overview of the main implementation relations on IOA and LTS.

An implementation relation is a relation that defines a notion of correctness between an implementation and a specification, to be more precise a model of the implementation and its specification (remember the test hypothesis). When the implementation relation holds, we say that the implementation conforms to the specification or, in other words, the specification is implemented by the implementation. Several implementation relations have been defined for the transition systems that were introduced in the previous section. When we treat an implementation relation that is also usable for one of the other types of transition systems we will use the general term *transition system* (where implementation and specification are the same type of transition system), otherwise we specify explicitly for which transition system the implementation relation is applicable.

We start with a rather weak relation: the trace inclusion pre-order. It expresses that one system is an implementation of the other if its set of traces is a subset of the set of traces of the specification.

Definition 2.4.1 [Trace inclusion] Let i and s be transition systems:

$$i \leq_{tr} s =_{\text{def}} \mathit{traces}(i) \subseteq \mathit{traces}(s)$$

Example 2.4.2 On the left-hand side of Figure 2.5 we see a specification of a coffee machine. We will reuse this coffee machine specification in other examples. It prescribes that after pressing a button at least twice we expect to observe either coffee or tea as output. On the right-hand side we see two implementations. The first implementation does not implement coffee as an output. It is still trace-inclusion-correct, because the set of traces of the implementation is a subset of the set of traces of the specification, even with the traces in $\mathsf{button}{\cdot}\mathsf{button}{\cdot}\mathsf{button}^*{\cdot}\mathsf{coffee}{\cdot}\mathsf{button}^*$ missing. We find trace inclusion not a very realistic implementation relation, because it also approves implementations that we find intuitively incorrect. For example, the implementation on the right only implements the pushing of the button, without serving any drink. This is correct according to the trace inclusion relation, because the set of traces $\mathsf{button}{\cdot}\mathsf{button}{\cdot}\mathsf{button}^*$ is a subset of the traces of the specification. □

The remainder of this section is organized as follows: we treat the main implementation relations defined for IOA in Section 2.4.1 and we treat the **ioco** theory and its implementation relations in Section 2.4.2.

2.4.1 Implementation relations defined for IOA

In this section we treat implementation relations that were originally defined for IOA: the fair and quiescent pre-order and the may and must pre-order. These implementation relations are pre-orders: the implementation and specification are the same type of transition system and the relation is reflexive and transitive. We start with the fair pre-order. This pre-order uses the notion of fair traces; it is only defined for IOA, because it requires a partitioning P of locally controlled actions.

Fair and Quiescent pre-order Based on the notion of fair traces, Lynch and Tuttle ([LT87]) define the fair pre-order. The combination of input-enabling and fairness guarantees that each implementation accepts all external stimuli, but provides output when the implementation must provide output.

Definition 2.4.3 [Fair pre-order] Given two IOAs i and s with the same external label sets, the fair pre-order is defined as:

$$i \leq_F s \Leftrightarrow \mathit{Ftraces}(i) \subseteq \mathit{Ftraces}(s).$$

Before we give an example of the fair pre-order we introduce a pre-order that is strongly related to the fair pre-order, namely the quiescent pre-order introduced by Vaandrager [Vaa91]. It uses the concept of quiescent traces.

Definition 2.4.4 [Quiescent pre-order] Given two transition systems i and s with the same external label sets, the quiescent pre-order is defined as:

$$i \leq_Q s \Leftrightarrow \mathit{traces}(i) \subseteq \mathit{traces}(s) \wedge \mathit{qtraces}(i) \subseteq \mathit{qtraces}(s).$$

The fair and quiescent pre-orders look much alike, but there are some important differences. The quiescent pre-order uses finite traces, whereas the fair pre-order includes infinite traces. The relation between the two pre-orders is easiest explained with an example (the example is reused with kind permission of Segala [Seg97]).

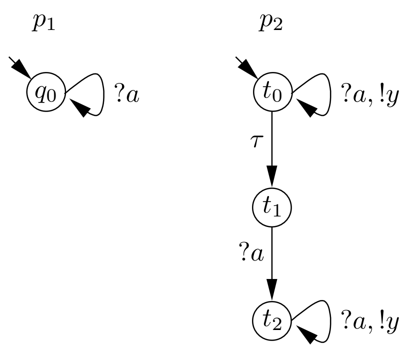


Figure 2.6: Quiescent versus fair pre-order, Example 2.4.5

Example 2.4.5 Figure 2.6 shows two IOA p_1 and p_2, a is an input action, y is an output action and τ is an internal action. The partition of locally controlled actions for both IOA is a single class $\{y, \tau\}$. We first treat the quiescent trace set. For p_1 we have $\mathit{traces}(p_1) = \mathit{qtraces}(p_1) = a^*$ (a set of zero or more occurrences of a). For p_2 we have $\mathit{traces}(p_2) = \{a, y\}^* \cdot a \cdot \{a, y\}^*$, $\mathit{qtraces}(p_2) = \{a, y\}^*$. Regarding the fair traces, for p_1 it is trivial that each finite sequence a^n is quiescent and therefore a fair trace. Also for p_2, the finite sequence a^n is a quiescent and fair trace. After looping n times in t_0 we move to t_1 via a τ transition. Therefore $p_1 \leq_Q p_2$. However, the sequence a^ω (infinite times a) is a fair trace of p_1 but not of p_2. The latter is because, to perform a^ω we are either infinitely often in t_0 or t_2. When we are infinitely often in t_0 we are not in a quiescent state, because τ is enabled. τ is in the partition of locally controlled actions, but does not occur infinitely often. When we are infinitely often in t_2, we are not in a quiescent state because y is enabled. But in this case y does not occur infinitely often in a^ω. Thus $p_1 \not\leq_F p_2$. □

MAY and MUST pre-order In this section we introduce MAY and MUST testing for systems with inputs and outputs. The method for comparing transition systems that was initiated by De Nicola and Hennessy is based on the observation of the interactions between a transition system and an external experimenter [DNH84]. The original theory was defined on labeled transition systems and did not take input and output actions into account. Segala compared the IOA model and the theory of testing of De Nicola and Hennessy and defined pre-orders directly on IOA ([Seg97]). The work in this section is based on this paper.

An experimenter e for a transition system p is a transition system that is compatible with p. Compatible means that the input actions of e are the output actions of p ($I_e = U_p$) and the output actions of e are the input actions of p, plus the experimenter has a unique (i.e., not in the other label sets) output action ω called the success action ($U_e = I_p \cup \{\omega\}$). The intuition behind the experimenter is that an implementation conforms to a specification if no external observer can see the difference. The experimenter e runs in parallel with p and synchronizes its output actions with input actions of p and vice versa (except ω). An experiment is an execution of $p\|e$ which is infinite or ends in a deadlocked state. We say that the experiment is successful if ω is enabled in at least one state of the experiment. If there is a successful experiment $p\|e$ we use the notation p MAY e. If every experiment $p\|e$ is successful we use the notation p MUST e. On this notion of MAY and MUST we can define pre-order relations. We will start with the MAY pre-order. We use the definition from Segala [Seg97]. This is a so-called *extensional* definition, because it refers to an external experimenter. The definitions so far are *intensional*, because they do not use an experimenter. Instead they focus on observable behavior.

Definition 2.4.6 [MAY pre-order] Let e, i, s be IOA:

$$i \leq_{\text{MAY}} s \Leftrightarrow \forall e : i \text{ MAY } e \Rightarrow s \text{ MAY } e$$

Definition 2.4.7 [MUST pre-order] Let e, i, s be IOA:

$$i \leq_{\text{MUST}} s \Leftrightarrow \forall e : i \text{ MUST } e \Rightarrow s \text{ MUST } e$$

It is also possible to define the MAY and MUST pre-orders in an intentional way, i.e., without referring to an external experimenter. For the MAY pre-order this is easy, as Hennessy has shown that the may pre-order and external trace inclusion are equivalent [Hen88].

Theorem 2.4.8 Let i, s be IOA:

$$i \leq_{\text{MAY}} s \Leftrightarrow \mathit{traces}(i) \subseteq \mathit{traces}(s)$$

□

For the must pre-order a bit more work is needed. Segala uses the following definition of the MUST relation [Seg97]. In it he uses a state property, similar to *init* called *wenabled* which is defined as: $\mathit{wenabled}(q) = \{a \in L \mid q \overset{a}{\Longrightarrow}\}$ where q is a state in the state set of the transition system.

Definition 2.4.9 [MUST$'$ relation] Given an IOA p, a set of states $Q \subseteq Q_p$ and a set of external actions $A \subseteq L_p$.

Q MUST$'$ $A \Leftrightarrow$

1. $A \cap I \neq \emptyset$, or
2. $\forall q \in Q : \mathit{wenabled}(q) \cap U \subseteq A \wedge \mathit{wenabled}(q) \cap A \neq \emptyset$

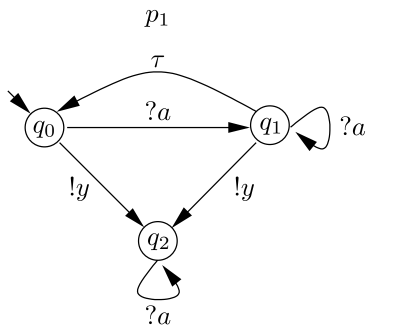


Figure 2.7: MUST testing example

Rule one of the MUST′ relation says that any transition system must perform its input actions: if there is an input action in A then the MUST′ relation automatically holds. Rule two says that an IOA decides which one of its output actions to perform (as longs as they are in A).

With this definition of the must relation on IOA we define the must pre-order in the following way.

Definition 2.4.10 [MUST′ pre-order] Let i, s be IOA with (external) label set L:

$$s \leq_{\text{MUST}'} i \Leftrightarrow \forall \sigma \in L^*, A \subseteq L : (s \textbf{ after } \sigma) \text{ MUST}' A \Rightarrow (i \textbf{ after } \sigma) \text{ MUST}' A$$

Example 2.4.11 We give an example of the MUST′ relation and show what this means for the MUST′ pre-order. Let us look at transition system p_2 (right-hand side) in Figure 2.7. It is straightforward to see that $\{t_0\}$ MUST′ $\{a, y\}$, because a is an input action (rule 1 of Definition 2.4.9). Likewise $\{t_0\}$ MUST′ $\{y\}$, because $init(q) = \{a, y\}$, therefore $init(q) \cap U = \{y\}$. But note that also $\{t_0\}$ MUST′ $\{x, y\}$, supposed that $U = \{x, y\}$. We leave it to the reader to verify that p_1 and p_2 are equivalent according to the MUST′ pre-order (and also to the quiescent pre-order; they are weakly bisimular). □

Segala has shown that this definition of the must pre-order is equivalent with (the inverse of) the quiescent pre-order, under the restriction that the transition systems are strongly converging (no τ-loops) and finitely branching (every state has finitely many transitions).

Theorem 2.4.12 Let i and s be finitely branching and strongly convergent IOA.

$$s \leq_{\text{MUST}'} i \Leftrightarrow i \leq_Q s.$$

□

2.4.2 IOCO based testing

In this section we introduce the implementation relations in the tradition of the **ioco** testing theory. The relations in this section are no pre-orders, because they take an IOLTS as a specification and assume the implementation to be an IOTS (we refer to the relations as implementation relations).

We treat the following implementation relations: $\leq_{iot}$, **ioconf**, $\leq_{ior}$ and **ioco**. For the implementation relations in this section we use the intentional definitions. The equivalence of our definitions with the original definitions is proved in [Tre96a].

Input-output testing relation The input-output testing relation tests with traces in L^*. This means that we can use any trace to test with, even if its behavior is not specified by the specification.

Definition 2.4.13 [Input output testing relation] Let $s \in \mathbf{IOLTS}(I, U), i \in \mathbf{IOTS}(I, U)$

$$i \leq_{iot} s =_{\text{def}} \forall \sigma \in L^* : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(s \textbf{ after } \sigma).$$

An implementation i is $\leq_{iot}$-correct with respect to a specification s if for all traces with which we test, the set of outputs of the implementation after such a trace is a subset of the set of outputs of the specification after the same trace. This means that we should not be able to observe different or more behavior from the implementation than from the specification.

Example 2.4.14 We illustrate the input output testing relation with the trace inclusion pre-order example in Figure 2.5 on page 28. The implementation i_1 is $\leq_{iot}$-correct with respect to specification s. We see that the implementation does not give coffee after pressing the button twice, so how can it be correct? Let us take a look at the definition of $\leq_{iot}$. The specification prescribes that the set of outputs after the trace $\mathsf{button}{\cdot}\mathsf{button} = \{\mathsf{coffee}, \mathsf{tea}\}$. When we take a look at i_1 we see that the set of outputs after $\mathsf{button}{\cdot}\mathsf{button} = \{\mathsf{tea}\}$. Because $\{\mathsf{tea}\} \subseteq \{\mathsf{coffee}, \mathsf{tea}\}$ it is correct behavior according to $\leq_{iot}$. The interpretation of choice in output is that we do not care which branch is implemented as long as at least one is. We can do the same analysis for implementation i_2. Here we find that after pressing the button twice i_2 does not give any output; it is quiescent. This means that $\mathit{out}(i_2 \textbf{ after } \mathsf{button}{\cdot}\mathsf{button}) = \{\delta\}$. This is not a subset of $\{\mathsf{coffee}, \mathsf{tea}\}$ and therefore $i_2 \not\leq_{iot} s$. □

ioconf relation The difference between **ioconf** and the input-output testing relation is that **ioconf** uses a different set of traces to test with, namely the set of all possible traces of the specification: $\mathit{traces}(s)$. This means that

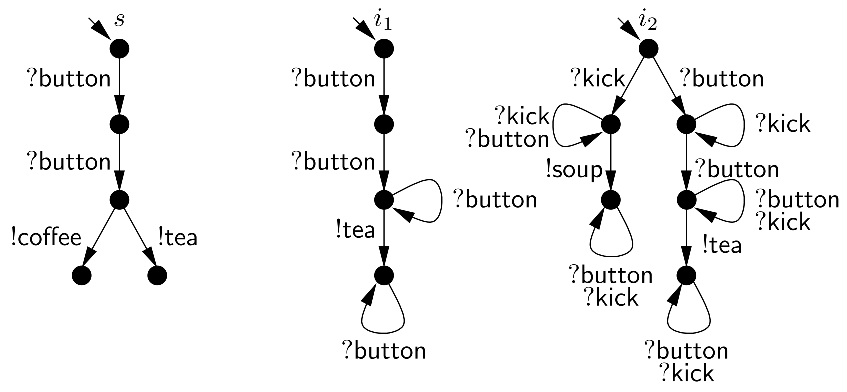


Figure 2.8: Example of **ioconf**

we will not test behavior that is not specified. One way to interpret this is as *implementation freedom*: "We do not know or care what the implementation does after an unspecified trace". The advantage is that we can test with incomplete specifications. Since $\mathit{traces}(s) \subseteq L^*$, the **ioconf** relation is weaker than the $\leq_{iot}$ relation.

Definition 2.4.15 [**ioconf**] Let $i \in \mathbf{IOTS}(I, U), s \in \mathbf{IOLTS}(I, U)$

$$i\ \mathbf{ioconf}\ s =_{\text{def}} \forall \sigma \in \mathit{traces}(s) : \mathit{out}(i\ \mathbf{after}\ \sigma) \subseteq \mathit{out}(s\ \mathbf{after}\ \sigma).$$

Example 2.4.16 In Figure 2.8 we illustrate the **ioconf** relation. i_1 is the same implementation as in the examples for external trace inclusion and $\leq_{iot}$. This implementation is correct under **ioconf**. This is easy to see, because the trace $\mathsf{button}\cdot\mathsf{button} \in \mathit{traces}(s)$; it is a trace of the specification and $\mathit{out}(i_1\ \mathbf{after}\ \mathsf{button}\cdot\mathsf{button}) = \{\mathsf{tea}\} \subseteq \mathit{out}(s\ \mathbf{after}\ \mathsf{button}\cdot\mathsf{button}) = \{\mathsf{coffee}, \mathsf{tea}\}$. Implementation i_2 introduces new behavior. When we kick the coffee machine it outputs soup. This behavior is nowhere to be found in the specification: the behavior of kicking the machine is underspecified. This kind of behavior would be a problem for $\leq_{iot}$ since it will test on all possible behavior of the label set: L^*. When we test the kicking of the machine with $\leq_{iot}$ we get the following result: $\mathit{out}(i_2\ \mathbf{after}\ \mathsf{kick}) = \{\mathsf{soup}\} \not\subseteq \mathit{out}(s\ \mathbf{after}\ \mathsf{kick}) = \emptyset$. In other words the implementation does not conform to the specification according to $\leq_{iot}$. This is the reason that with $\leq_{iot}$ we need an input-complete specification, otherwise the relation will never hold after an unspecified input action. **ioconf** does not have this restriction, because it will only test behavior that is specified. Since $\mathsf{kick} \notin \mathit{traces}(s)$, we will not test this behavior. Because all the other behavior of i_2 is identical to i_1 we have i_2 **ioconf** s. □

Input-output refusal relation The input-output refusal relation, uses L_δ^*

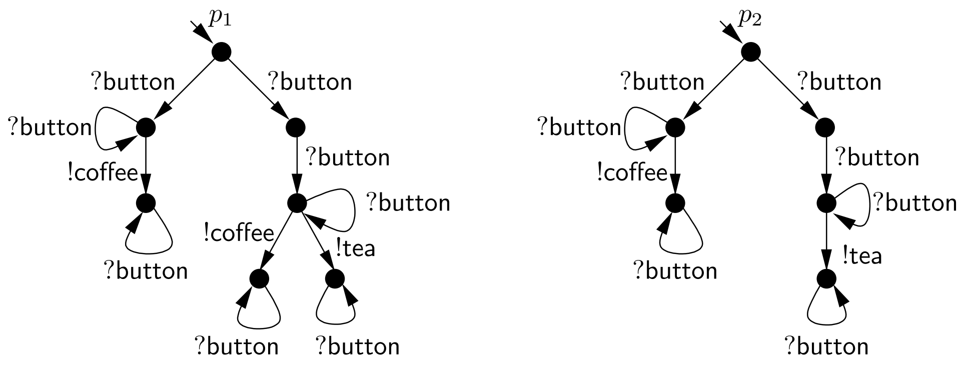


Figure 2.9: Comparison between **ioco** and other relations

as the set of traces to test with. This makes it possible to continue testing after the observation of quiescence. With the previous implementation relations it was only possible to observe quiescence at the end of a trace. Quiescence can be seen as *refusal* to do an output action, hence the name of the implementation relation. Because we test with the entire label set, it only makes sense to test with complete specifications as was illustrated for $\leq_{iot}$ in Example 2.4.16. Again, the correctness criterion is that an implementation does not show more output behavior than is allowed by the specification.

Definition 2.4.17 [Input output refusal] Let $i \in \mathbf{IOTS}(I, U)$ and $s \in \mathbf{LTS}(I, U)$

$$i \leq_{ior} s =_{\text{def}} \forall \sigma \in L_\delta^* : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma).$$

We give an example of the $\leq_{ior}$ implementation relation together with **ioco**, because these two implementation relations are closely related.

ioco relation The **ioco** testing theory is named after its implementation relation **ioco**. **ioco**, like $\leq_{ior}$, allows us to continue testing after the observation of quiescence. In order to be able to test with incomplete specifications we restrict the set of traces to the suspension traces of the specification. This set is smaller than the set of traces of $\leq_{ior}$. In other words, **ioco** is a weaker implementation relation than $\leq_{ior}$.

Definition 2.4.18 [**ioco**] Let $i \in \mathbf{IOTS}(I, U), s \in \mathbf{LTS}(I, U)$.

$$i \textbf{ ioco } s =_{\text{def}} \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma).$$

Example 2.4.19 We illustrate the $\leq_{ior}$ and **ioco** implementation relations in Figure 2.9 in an example from Tretmans [Tre96a]. We see two IOTSs

p_1 and p_2 that model a coffee machine with peculiar behavior. p_1 models a machine where after pressing a button once, we get either coffee or nothing (quiescence). If we got nothing and we press the button again we get either tea or coffee. p_2 models an almost identical machine, except that after we press the button again after obtaining nothing after the first button press we will only get tea (so no coffee). If p_1 is the implementation and p_2 the specification we see that $p_1 \leq_{iot} p_2$ and p_1 **ioconf** p_2, but not $p_1 \leq_{ior} p_2$ and not p_1 **ioco** p_2. Let us begin with **ioco**, **ioco** can see the difference between the transition systems because of the following trace. After button·δ·button we will observe tea and coffee for p_1, whereas p_2 prescribes that only tea is allowed. The same holds for $\leq_{ior}$ since it is also capable of this same test case. However $\leq_{iot}$ and **ioconf** are not capable of observing quiescence during testing and can therefore not tell the difference between the trace button·button·coffee in the left branch of the transition system or in the right branch of the transition system. In other words, they are not powerful enough to see the difference.

When we take p_2 as the implementation and p_1 as the specification we see that all implementation relations identify the implementation as correct. This is logical since the only difference between p_1 and p_2 is that p_2 does not offer the possibility of coffee in the right branch. This is correct, since the specification p_1 gives the choice between implementing either one (or both).

□

The structure of the definitions of the implementation relations in this chapter makes it easy to compare them. The only variable is the set of traces over which we test. The proposition below (by Tretmans [Tre96a]) expresses the relation between the implementation relations. The rationale is that: $traces(s) \subseteq Straces(s) \subseteq L_\delta^*$ and $traces(s) \subseteq L^* \subseteq L_\delta^*$.

Proposition 2.4.20 Comparison of expressiveness of the implementation relations.

$$\leq_{ior} \subset \left\{ \begin{array}{c} \leq_{iot} \\ \mathbf{ioco} \end{array} \right\} \subset \mathbf{ioconf}$$

□

This seems an appropriate place to mention the relation with MAY and MUST pre-orders. Tretmans showed that $\leq_{iot}$ is equivalent with the quiescent pre-order, and thus equivalent to the MUST pre-order [Tre96a]. The MAY pre-order is equivalent with trace inclusion [Hen88].

Introspection When we look at the history of **ioco**, we can make the following observations. We find the **ioco** definition rather elegant and easy to read because of its intensional definition, i.e., a definition without external observers. On top of this the relation appeals to an intuitive correctness notion by comparing the output of the implementation with the output prescribed by the specification after a defined trace (by the specification). Compare this with the other relations that look at some form of trace inclusion. **ioco** assumes implementations to be input-enabled. We find this a realistic assumption, since in many cases a tester can provide an implementation with stimuli, whether the implementation expects it or not. We do feel that the consequences of input-enabled implementations versus non-input-enabled specifications are a bit hidden in the theory. Especially the notion of underspecification of input actions in the specification needs more attention as we will show in Chapter 3.

2.5 Testing transition systems

In the previous sections, we have discussed several I/O-models and implementation relations. In this section, we introduce two more concepts of the conformance testing framework, namely test derivation and test execution, targeted specifically for **ioco**. We will show the relation between **ioco** and test generation and execution. We start with the introduction of test-cases.

Test-cases A test-case is a specification of the experiment that an experimenter wants to conduct on an implementation. A test-case can be modeled by an IOLTS. We add a couple of restrictions to the behavior of a test-case. To guarantee that a test-case finishes in finite time it should have finite behavior. Furthermore to ensure maximal control over the testing process we do not allow non-deterministic behavior. We also do not allow choice between multiple input actions and between input actions and output actions. This implies that a state in a test-case is either a terminal state, or a state that offers exactly one input to the implementation or accepts all outputs of the implementation. To give a verdict over the success of the test-case we label terminal states with **pass** and **fail**. These restrictions are formally expressed in the following definition of a test-case. Note that a test-case could in principle be defined without these restrictions. It could be an arbitrary LTS that synchronizes on the actions of the implementation under test. The definition we introduce here has shown to be both theoretically and practically useful.

Definition 2.5.1 [Test-case] A test-case $t = \langle Q,\ R,\ S,\ T,\ \mathsf{start},\ \mathsf{Pass},\ \mathsf{Fail}\rangle$ over a set of *stimuli* $S \subseteq L$ and *responses* $R \subseteq L_\delta$ $(S \cap R = \emptyset)$ is an acyclic IOLTS $\langle Q, S, R, T, \mathsf{start}\rangle$ such that:

- t is deterministic and has finite behavior.
- $\mathsf{Pass}, \mathsf{Fail} \subseteq Q$ with $\mathsf{Pass} \cap \mathsf{Fail} = \emptyset$ and $\forall q \in \mathsf{Pass} \cup \mathsf{Fail} : \mathit{init}(q) = \emptyset$.

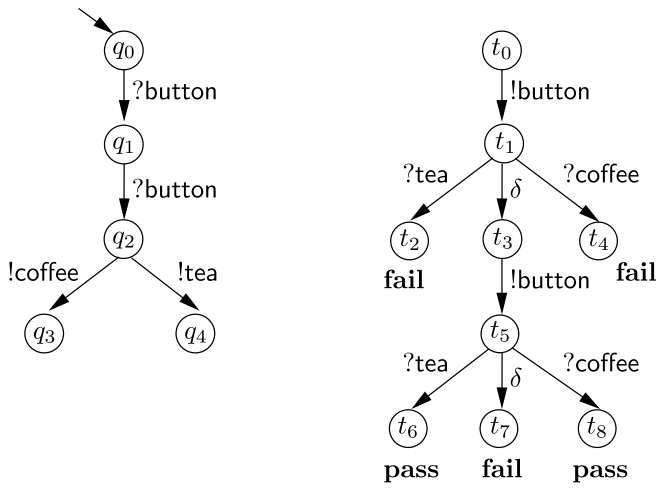


Figure 2.10: Example of a test-case

- For any state $q \in Q$ of the test-case, if $q \notin \mathsf{Pass} \cup \mathsf{Fail}$ then either $init(q) = \{a\}$ for some $a \in S$, or $init(q) = R_\delta$.

The class of test-cases over R and S is denoted as $\mathbf{TEST}(R, S)$. A test suite T is a set of test-cases: $T \subseteq \mathbf{TEST}(R, S)$.

We will use the terms *stimulus* for an output of the test-case (i.e., an input of the implementation) and *response* for an input of the test-case (i.e., an output of the implementation). We deviate from Tretmans' definition of a test-case in that we denote the observation of quiescence with δ instead of θ and that we use sets of pass and fail states instead of one pass state and one fail state. In figures we denote a state in Pass by writing **pass** underneath it, likewise **fail** for states in Fail.

For a test-case t and an arbitrary trace σ we write $t \xrightarrow{\sigma}$ **pass**, if and only if there exists a state $q \in \mathsf{Pass}$ such that $t \xrightarrow{\sigma} q$. In a similar way we use the notation $t \xrightarrow{\sigma}$ **fail**

Example 2.5.2 On the right-hand side of Figure 2.10 we show an example of a test-case. With this test-case we can test the coffee machine specified on the left-hand side. We see that the test-case starts with the stimulus button in state t_0. In state t_1 it makes an observation. The specification prescribes that there should be no output. So, if we observe coffee, or tea we add the end state to the set of fail states Fail, like in t_2 and t_4. If we observe quiescence we continue testing. Again we stimulate the implementation with button and arrive in state t_5. Now the specification prescribes that we can observe coffee or tea as valid responses. This means that the observation of quiescence leads to state t_7 in Fail. If we observe coffee or tea we stop testing: the states t_6 and t_8 are states in Pass. □

Test execution The execution of a test-case on (a model of) an implementation is modeled by synchronous parallel composition (denoted by $\|$) of the test-case with the implementation under test. This means that inputs of the test-case synchronize on outputs of the implementation and vice versa. In case of quiescence, the test-case synchronizes on δ. The execution continues until the test-case reaches a pass or fail state. Because of the special structure of a test-case and input-enabledness of the IUT, we are sure that the test-case will always eventually reach a pass or fail state. An implementation passes the test if the test-case ends in a pass state, if it ends in a fail state we say that the implementation fails the test-case. This means that we have found a deviation between the model and the implementation; a potential error. Because an implementation can have non-deterministic behavior, different executions with the same test-case can lead to different terminal states (and possibly different verdicts). Therefore, an implementation passes a test-case if *all* possible test executions lead to the verdict pass.

Definition 2.5.3 Let $t \in \mathbf{TEST}(R, S)$ and $i \in \mathbf{IOTS}(I, U)$.

1. An execution of a test-case t with an implementation i is a trace of the synchronous parallel composition $t\|i$ leading to a terminal state of t:

   σ is a test execution of t and i iff $\exists i' : t\|i \overset{\sigma}{\Longrightarrow} \mathbf{pass}\|i'$ or $t\|i \overset{\sigma}{\Longrightarrow} \mathbf{fail}\|i'$.

2. Implementation i passes test-case t if all their test executions lead to the pass-state of t:

   $$i \ \mathbf{passes}\ t =_{\text{def}} \forall \sigma \in L_\delta^*, \forall i' : t\|i \overset{\sigma}{\not\Longrightarrow} \mathbf{fail}\|i'.$$

3. An implementation i passes test suite T if it passes all test-cases in T:

   $$i \ \mathbf{passes}\ T =_{\text{def}} \forall t \in T : i \ \mathbf{passes}\ t.$$

   If i does not pass the test suite, it fails: i **fails** $T =_{\text{def}} i \ \cancel{\mathbf{passes}}\ T$.

Note that the label-set of the parallel composition $t\|i$ has the signature: $I_{t\|i} = (R_t \cup I_i)\backslash G$ and $U_{t\|i} = S_t \cup U_i$ with $G = (R_t \cap U_i) \cup (S_t \cap I_i)$. In general, a test-case has the inverse label-set of the IUT: $S = I$ and $R = U$ (a stimulus/input for the test-case is an output from the IUT and a response/output from the test-case is an input for the IUT). This results in the interesting signature $I_{t\|i} = \emptyset$ and $U_{t\|i} = I \cup U$

As explained in Section 2.2, completeness, soundness and exhaustiveness are important properties of test-cases/test suites. Soundness expresses that if an implementation fails a test-case, there really is an error in the implementation according to the specification (i.e., not an error in the test-case).

An exhaustive test suite means that if there is an error in the implementation, there is a test-case in the test suite to detect it. Note that in practice exhaustiveness often means a (practically) infinite test suite. One loop in the specification makes an exhaustive test suite infinite, because an error can occur after an arbitrary number of iterations of the loop. An exhaustive test suite needs to take this infinite behavior into account. We call the combination of soundness and exhaustiveness *completeness*. Below we give the formal definitions.

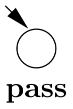
Definition 2.5.4 Let s be a specification and T a test suite then:

T is complete	$=_{\text{def}}$	$\forall i : i \textbf{ ioco } s \Leftrightarrow i \textbf{ passes } T$
T is sound	$=_{\text{def}}$	$\forall i : i \textbf{ ioco } s \Rightarrow i \textbf{ passes } T$
T is exhaustive	$=_{\text{def}}$	$\forall i : i \textbf{ ioco } s \Leftarrow i \textbf{ passes } T$

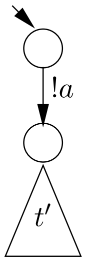
Test derivation We finish the instantiation of the test framework with test derivation (also called test generation). A minimal requirement for a test derivation algorithm is that it derives sound test-cases. As mentioned above, exhaustiveness is an interesting requirement for a test derivation algorithm, because of the possibly infinite size of these test suites. Therefore some people argue to a priori choose to derive a finite test suite. This seems to have some practical advantage as the derived test suites are always finite and, if well done, we know what test-cases are in the test suite (and which ones are not). The downside of this choice is that there are non-conforming implementations that we cannot detect because we are unable to derive the necessary test-cases that detect the errors. We find this an unacceptable property of a test-case derivation algorithm. Our point of view is that in principle we should be able to find all possible errors in an IUT and that for practical reasons we may have to limit the amount of derived test-cases (but not the other way around).

It turns out that a relatively simple algorithm can potentially produce a complete test suite for **ioco**. Test generation algorithms for the other implementation relations can be made in a similar way. For the completeness proof we refer to Tretmans [Tre96a]. In the definition of the test derivation algorithm we have chosen to use the process notation introduced in Section 2.3.2 to make it easier to read. We added pictures to represent the way a test-case is built up (the pictures are fragments of test-cases) from the behavioral expressions. Furthermore we give an example of the derivation of a test-case after the definition.

Definition 2.5.5 Let $s \in \textbf{IOLTS}(I, U)$ be a specification. Let S be a non-empty set of states, with initially $S = \{\mathsf{start}\}$. A test-case $t \in \textbf{TEST}(I, U_\delta)$ is obtained from S by a finite number of recursive, non-deterministic applications of one of the following three choices:

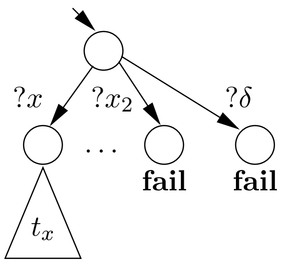
1.


$t :=$ **pass**
The test-case with only the state **pass** is always a sound test-case. This rule stops the recursion in the algorithm.

2.


$t := a; t'$ where $a \in I_s$ and S **after** $a \neq \emptyset$. t' is obtained by recursively applying the algorithm for $S' = S$ **after** a.

This step of the algorithm adds a stimulus a to the test-case. After applying a, the test-case behaves as t' which is obtained by applying the test derivation algorithm recursively to S'. t' is depicted as an abstract sub-tree (triangle) in the figure.

3.


$$
\begin{aligned}
t := \quad & \Sigma\ \{x; \textbf{fail} \mid x \in U, x \notin out(S)\} \\
+ \quad & \Sigma\ \{\delta; \textbf{fail} \mid \delta \notin out(S)\} \\
+ \quad & \Sigma\ \{x; t_x \mid x \in U, x \in out(S)\} \\
+ \quad & \Sigma\ \{\delta; t_\delta \mid \delta \in out(S)\}
\end{aligned}
$$

t_x and t_δ are obtained by recursively applying the test derivation algorithm for $S' = S$ **after** x, S **after** δ, respectively.

This step of the algorithm adds expected outputs to the test-case. If the output is incorrect according to the specification we add a transition with the output to a fail state, thus ending that part of the test-case. For outputs that are allowed, we continue the test derivation with t_x, t_δ respectively.

Example 2.5.6 We illustrate the test derivation algorithm with our coffee machine specification shown on the left-hand side in Figure 2.10 on page 38. We will show how to derive a test-case with the algorithm. We derive the same test-case as the one used in Figure 2.10 on page 38. When we start, the set S consists of only the start state q_0 of our specification. We non-deterministically choose one of the three rules of the test derivation algorithm. We start by applying rule 2 of the test derivation algorithm and apply the input button. This is possible, since q_0 **after** button $= \{q_1\}$ ($\neq \emptyset$). The result is $t_0 = \mathsf{button}; t_1$ (the transition $(t_0, !\mathsf{button}, t_1)$ in our test-case). The set S is updated to $S = \{q_1\}$. We choose to observe responses from the implementation under test: we apply rule 3. There are three possible responses: tea, coffee and quiescence. We compute $out(q_1) = \{\delta\}$ of the specification. This means that the only allowed output is quiescence. By applying our algorithm we obtain: $t_0 = \mathsf{button}; (\mathsf{tea}; t_2 + \delta; t_3 + \mathsf{coffee}; t_4)$ with

t_2 and t_4 fail states. In other words, we add a transition with tea to a fail-state $(t_1, ?\mathsf{tea}, t_2)$ and a transition with coffee to a fail state $(t_1, ?\mathsf{coffee}, t_4)$. For the allowed response δ we add the transition (t_1, δ, t_3). We update S with S **after** $\delta = \{q_1\}$. We again apply the stimulus button which results in the transition $(t_3, \mathsf{button}, t_5)$ and $S = \{q_2\}$ (we skip the behavior expression, to keep the example terse). For rule 3 there are two options, either the response coffee or the response tea, since $out(q_2) = \{\mathsf{coffee}, \mathsf{tea}\}$. We add the transitions $(t_5, ?\mathsf{tea}, t_6)$, $(t_5, ?\mathsf{coffee}, t_8)$ and (t_5, δ, t_7) where t_7 is a fail-state. At this point the specification has two possible paths to continue with. For the "tea" path we update S with $\{q_4\}$ and for the "coffee" path we update S with $\{q_3\}$. We can in principle continue forever with alternating between rule 2 and 3 of the test derivation algorithm until we reach a final state in the specification or until we want to stop. In our case the specification has reached a final state and we can apply rule 1 to stop the recursion. This transforms states t_6 and t_8 into **pass**-states. □

2.6 Conclusion and introspection

The purpose of this chapter is to put up the necessary scaffolding for the rest of the thesis. We introduced a conformance testing framework for LTS-based testing with inputs and outputs and we filled in and explained the parts and pieces of this framework. To recapitulate, the *conformance relation* important for this thesis is **ioco**. **ioco** assumes the *specification* to be an IOLTS and it assumes that the *implementation* can be modeled as an IOTS. We have shown an algorithm to derive *test-cases* from a specification. The actual testing, consists of executing the test-cases against the implementation. The testing results in a *verdict*. If the implementation shows behavior that we expect, based on the specification, the implementation passes the test-case, otherwise it fails.

We like the **ioco** theory because of its simplicity, elegance and lack of restrictions. For example, compared to FSM-based testing, **ioco** allows specifications to be non-deterministic, infinite and partially specified. There are two characteristics of the **ioco** theory in particular, that we like to emphasize. The first one is that it uses LTS-based specifications, the other is that the theory assumes implementations to be input-enabled.

We find LTS-based specifications important, because it gives the theory a powerful, yet simple formalism with a clear semantics. For example, input and output actions are separated. This may seem trivial, but for example the FSM-based approach combines input and output actions in one transition. In our opinion testing is about experimenting with the implementation by providing stimuli and observing responses and we find it important that the specification formalism supports this. The separation of input and output transitions is very important, if not a must, for our work on action refinement (Chapter 5). Another characteristic of an LTS that we find important is that it has well defined operations defined on it. For example an operation that is important for this thesis is compositionality, in particular parallel composition. We treat compositionality and the **ioco** theory in the next chapter. Input-enabledness of implementations reminds us that implementations are special. In general a tester can always send a stimulus to the implementation under test. Possible exceptions are graphical interfaces, or systems with a physical interface. When a button is not available on the screen we cannot click it and if a coin slot of a vending machine is blocked, we cannot enter a coin. The next chapter will show that compositionality and input-enabledness are important properties in the **ioco** theory.

While the **ioco** theory has pleasant characteristics, there is certainly room for improvement. It would be very nice if the strong convergence restriction on transition systems can be lifted. Divergence is considered to be a bad property of a specification, however there is a category of perfectly normal systems that are divergent by nature. For example systems that send a message every x time units, with x a certain amount of time (for example 2 seconds). Hiding this message creates a divergent system. We find that it is up to the modeler to decide if convergence is a necessary property. Related to convergence is the observation of quiescence. For practical testing, quiescence is implemented by observing responses until a certain time-out. This means that practically, quiescence is time-related, real quiescence cannot be observed from the outside. We think it is a good idea to integrate this notation of time explicitly into the theory. This can for example be done by introducing a notion of fairness on output actions. If the implementation can perform an output according to the specification, it should be visible after a certain amount of time. This also solves the convergence restriction: infinite τ-loops cannot block outputs forever.

Two other, probably related, issues are the role and definition of test-cases and the test-case generation algorithm. On the intuitive level traces, and test-cases are similar, closely related concepts (a test-case can even be represented as a set of traces): a test-case tests if the traces of an implementation are conform the traces prescribed by the specification. Look for example at the similarity in the proofs for traces and test-cases in Chapter 5. Also in practice, hand-made test-cases often do not look like the tree-like structures of the **ioco** theory. They look more like traces or test-purposes [dVT01]. It feels like the theory could be improved in this area.

The test case generation algorithm of Section 2.5 describes a way to construct test-cases; it is constructive. It would be nice to have an explicit characterization of a test-suite needed for conformance testing. Then we can show that the test-case generation algorithm fulfills the characterization. Starting with the algorithm seems the wrong order.

On the positive side, this means that there is still work to do. Luckily the **ioco** community is relatively active. Furthermore it has tool-support.

We find it important to test theory in practice (many test theories and test-relations exist only on paper). The **ioco** theory is implemented in some test-tools, like TorX [dVT98] and TGV [FJJV96, FJJV97, CJRZ02]. TGV uses an implementation relation quite similar to **ioco** (the interpretation of quiescence differs in that τ-loops are also interpreted as quiescence). Lately the **ioco**-theory has been extended with time [BB04] and hybrid models [vO06] and the original TorX tooling is rewritten into the more portable Java framework [JTo, Bel10].

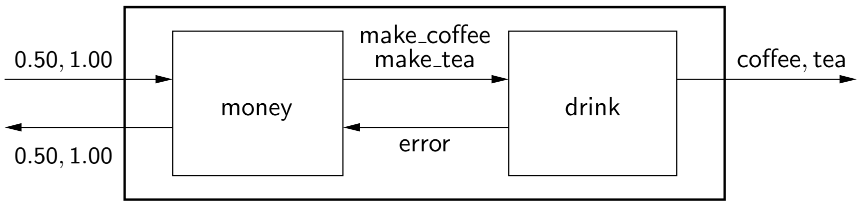
Chapter 3

Compositional testing with ioco

In this chapter we show that standard **ioco** cannot be used for compositional testing. We give an analysis why this is the case and provide some solutions. We show that when specifications are modeled by IOTSs (as opposed to LTSs), compositional testing works fine. We introduce *demonic completion*: a way to transform an LTS to an IOTS (in order to enable compositional testing) and based on this notion, we introduce a new implementation relation **uioco**. While **uioco** solves some of the compositionality problems of **ioco**, it unfortunately is not compositional itself. Finally we introduce a new parallel composition operator that supports compositional testing. Furthermore we show that the results for compositional testing are also applicable for testing in context. The contents of this chapter are based on our work on compositional testing [vdBRT04].

3.1 Introduction

COMPOSITIONAL TESTING is the topic of this chapter: the testing of communicating components that together form a larger system. Compositional testing in this sense is known under different names. To name two: *component based testing*, i.e., integration testing of components that have already been tested separately; and *interoperability testing*, i.e., testing if systems from different manufacturers, that should comply with a certain standard, work together; for example when testing if GSM mobile phones from different manufacturers can communicate with each other. The central question in this chapter is: "what can be concluded from the individual tests of the separate components, and what should be (re)tested on the integration or system level"? We will show that in the traditional **ioco** theory (i.e., before our work) it is unclear what the relationship between the correctness of the components and the integrated system is.



$$s_{\mathsf{cof}} = \textbf{hide}\ \{\mathsf{make\_coffee}, \mathsf{make\_tea}, \mathsf{error}\}\ \textbf{in}\ s_{\mathsf{mon}} \| s_{\mathsf{drk}}$$
$$i_{\mathsf{cof}} = \textbf{hide}\ \{\mathsf{make\_coffee}, \mathsf{make\_tea}, \mathsf{error}\}\ \textbf{in}\ i_{\mathsf{mon}} \| i_{\mathsf{drk}}$$

Figure 3.1: Architecture of coffee machine in components

Another scenario, with similar characteristics, is *testing in context*. This refers to the situation that a tester can only access the implementation under test through a *test context* [ISO96, JJTV99, PYVB96]. As a consequence the tester can only indirectly observe and control the IUT via the test context. This makes testing weaker, in the sense that there are fewer possibilities for observation and control of the IUT. With testing in context, the question is whether faults in the IUT can be detected by testing the composition of IUT and test context, and whether a failure of this composition always indicates a fault of the IUT. This question is the converse of compositional testing: when testing in context we wish to detect errors in the IUT — a component — by testing it in composition with the test context, whereas in compositional testing we judge correctness of the integrated system from conformance of the individual components.

Compositional testing and testing in context are issues in the testing theory in general, and in the **ioco**-theory in particular. For the testing theory based on Finite-State-Machines (FSM) this issue has been studied in [PY97], where the authors use so called communicating FSMs. To put it in **ioco** terms we investigate in this chapter if **ioco** correctness of the components implies **ioco** correctness of the entire system (formed by the components). We focus on two operations that are important in compositional testing: parallel composition and hiding. With parallel composition we can form a system of communicating components. Hiding makes it possible to hide communication between components.

Example 3.1.1 We illustrate compositional testing with a coffee machine that consists of a "money component" (money) and a "drink component" (drink). money handles the inserted coins and drink takes care of preparing and pouring the drinks. In Figure 3.1 we see the coffee machine consisting of the components money and drink. The purpose of the outer border is to identify the individual components and the composed system.

The money component accepts coins of €1 and of €0.50 as input from the environment. After insertion of a €0.50 coin (or a €1 coin), the money component orders the drink component to make tea with the $\mathsf{make\_tea}$ command (or coffee with the $\mathsf{make\_coffee}$ command). If something goes wrong in the drink making component the money component receives an error signal and gives the inserted money back.

The drink component interfaces with the money component and the environment. If the drink component gets the $\mathsf{make\_tea}$ command (or the $\mathsf{make\_coffee}$ command) it outputs tea (or coffee, respectively) to the environment. If anything goes wrong in the drink making process, the component gives an error signal.

The coffee machine is the parallel composition of the components money and drink, in which the commands $\mathsf{make\_coffee}, \mathsf{make\_tea}$ and error are hidden. One can think of the parallel composition as establishing the connection between the money component and the drink component, whereas hiding means that the communication between the components is not observable anymore; only communication with the environment can be observed. This means that to the environment the coffee machine is a black box: all communication inside the box is hidden. The only observable actions are $\mathsf{€1.00}, \mathsf{€0.50}, \mathsf{coffee}$ and tea. When we look inside the box we see the communicating components money and drink. We denote the specification of the money component and drink component as s_{mon} and s_{drk}, respectively. Likewise we use i_{mon} and i_{drk} for the implementations. We use s_{cof} and i_{cof} to denote the coffee machine specification and implementation, respectively.

□

In terms of the coffee machine example, the question we investigate in this chapter is: "If the money component and drink component have been tested to be correct implementations, may we conclude that their integration is also correct?". In Section 3.2 we extend our coffee machine example.

The main result of this chapter is that we show that **ioco** can be used for compositional testing. Either by using IOTS specifications, or by using our adapted parallel composition operator. We show that underspecification of input actions is a problem when testing communicating components with the **ioco** theory. This idea is new for LTS testing. It is inspired by [DNS95] and similar work done in FSM testing [PBD94]. We show that the results on compositional testing are also applicable for testing in context.

Overview In Section 3.2 we extend our coffee machine example and we formalize the problems of compositional testing and testing in context. Section 3.3 studies preservation of **ioco** for parallel composition and hiding. Section 3.4 discusses underspecification. It treats three ways to tackle underspecification of input actions in LTSs: one is to remove underspecification

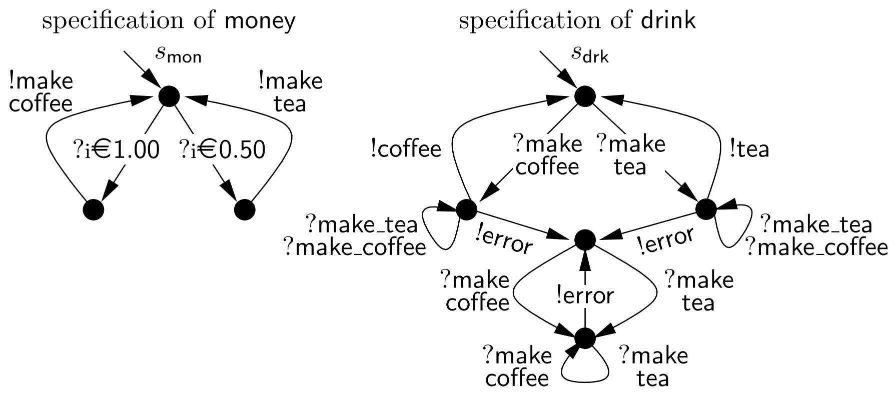


Figure 3.2: Specification of money and drink components as LTSs

entirely by making an LTS specification input enabled, another is to change the definition of **ioco** and the third is to change the definition of the parallel composition operator. In Section 3.5 we show that the results for compositional testing are also applicable for testing in context. Section 3.6 concludes with some final remarks and an assessment of the results.

3.2 Approach

We study systems that consist of communicating components. These components can be tested individually and while working together (in the case of testing in context the components are the IUT and its test context). The behavior of such a system is described by the parallel composition of the individual transition systems. Output actions of one component that are in the input label set of another component are *synchronized*, resulting in a single, output transition of the overall system. Actions of a component that are not in the label set of another component are not synchronized, resulting in a single observable transition of the overall system (see also Section 2.3.2). In case the synchronized action cannot be observed anymore, for example because the interface between the components is hidden, we apply hiding after the parallel composition, rendering the synchronized action to an internal action.

In Example 3.1.1 we illustrated compositional testing with a coffee machine. The following example treats compositional testing in more detail by looking at the transition systems of the components in our coffee machine example.

Example 3.2.1 Figure 3.2 shows the LTSs of the specifications of the money component (s_{mon}) and the drink component (s_{drk}). s_{mon} outputs a make_coffee command after insertion of i€1.00 and a make_tea command

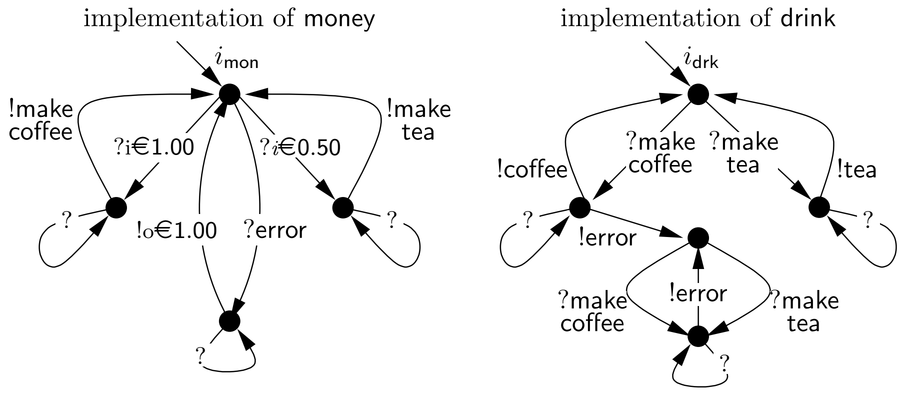


Figure 3.3: Implementation of the money and drink components as IOTSs

after insertion of i€0.50. The money labels occur as input and output actions. Because input and output label-sets are supposed to be disjoint we add ‘i’ and ‘o’ to the money labels. The label sets of s_{mon} are $I_{\mathsf{mon}} = \{$i€0.50, i€1.00, error$\}$ and $U_{\mathsf{mon}} = \{$make_coffee, make_tea, o€0.50, o€1.00$\}$. The drink component outputs coffee after it receives the make_coffee command and tea after it receives the make_tea command. When something goes wrong in the coffee or tea making process, for example when the machine is out of coffee, it outputs an error signal (note that the money component is underspecified for error). The drink component cannot recover from an error state and while in the error state it cannot produce tea or coffee. The drink component has the following label sets: $I_{\mathsf{drk}} = \{$make_coffee, make_tea$\}$ and $U_{\mathsf{drk}} = \{$coffee, tea, error$\}$. Figure 3.3 shows implementation models of the money component, i_{mon}, and the drink component, i_{drk} (as IOTSs). We have used transitions labeled with ‘?’ as an abbreviation for all the non-specified input actions from the alphabet of the component. The label sets for the implementations are the same as the label sets for the specifications of the components.

In the implementations we choose to improve upon the specification, by adding functionality. This is possible since **ioco** allows partial specifications. Implementers are free to make use of the underspecification. The extra functionality of i_{mon} compared to its specification s_{mon} is that it can handle error signals: it reacts by returning €1.00. i_{drk} is also changed with respect to its specification s_{drk}: making tea never produces an error signal. Since implementations are input enabled, we have chosen that all non specified inputs are ignored, i.e., the system remains in the same state.

The result is that we have implementations of the components that are **ioco** correct with respect to their specifications: i_{mon} **ioco** s_{mon} and i_{drk} **ioco** s_{drk}. The question now is whether the integrated implementation, as depicted by i_{cof} in Figure 3.1 on page 46, is also **ioco** correct with respect to the integrated specification s_{cof}. We discuss this in Section 3.3. □

3.3 Central questions in compositional testing

We paraphrase the question of compositional testing, discussed in the introduction, as follows: "Given that the components p and q have been tested to be **ioco**-correct (according to their respective specifications), may we conclude that their integration is also **ioco**-correct (according to the integrated specification)?". If the component specifications are LTSs, the component implementations are modeled by IOTSs, and their integration by parallel composition followed by hiding, this boils down to the following questions in our formal framework (we introduced parallel composition and hiding in Section 2.3.2) where $i_k \in \mathbf{IOTS}(I_k, U_k)$ and $s_k \in \mathbf{LTS}(I_k, U_k)$ for $k = 1, 2$, with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$:

Q1: Given i_k **ioco** s_k for $k = 1, 2$, is it the case that $i_1 \| i_2$ **ioco** $s_1 \| s_2$?

Q2: Given i_1 **ioco** s_1, is it the case that (**hide** V **in** i_1) **ioco** (**hide** V **in** s_1) for arbitrary $V \subseteq U_1$?

If the answer to both questions is *yes*, then we may conclude that **ioco** is suitable for compositional testing as stated in the following conjecture.

Conjecture 3.3.1 If $i_k \in \mathbf{IOTS}(I_k, U_k)$ and $s_k \in \mathbf{IOLTS}(I_k, U_k)$ for $k = 1, 2$ with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ and $V = (I_1 \cap U_2) \cup (U_1 \cap I_2)$, then

$$i_1 \ \mathbf{ioco}\ s_1 \wedge i_2 \ \mathbf{ioco}\ s_2 \Rightarrow (\mathbf{hide}\ V\ \mathbf{in}\ i_1 \| i_2)\ \mathbf{ioco}\ (\mathbf{hide}\ V\ \mathbf{in}\ s_1 \| s_2)\ .$$

We will show in the remainder of this section that the answer to Q1 and Q2 is *yes* if s_1 and s_2 are specified as IOTSs (i.e., completely specified for their input label sets). Otherwise the answer is no.

3.3.1 Parallel composition

In the following example we show that *if* we have two **ioco**-correct component implementations, *then* the implementation does *not* remain correct after parallel composition of the components.

Example 3.3.2 Regard the LTSs in Figure 3.4 on the next page. On the left-hand side we show the specifications and on the right-hand side the corresponding implementations. The models have the following label sets: $s_1 \in \mathbf{LTS}(\{x\}, \emptyset), i_1 \in \mathbf{IOTS}(\{x\}, \emptyset), s_2 \in \mathbf{LTS}(\emptyset, \{x\})$ and $i_2 \in \mathbf{IOTS}(\emptyset, \{x\})$. We see that s_1 and i_1 can perform one input x, after that s_1 is underspecified and i_1 continues to accept x actions as it is input enabled. s_2 and i_2 are identical and output two x actions (the suspension traces of s_1 are given by

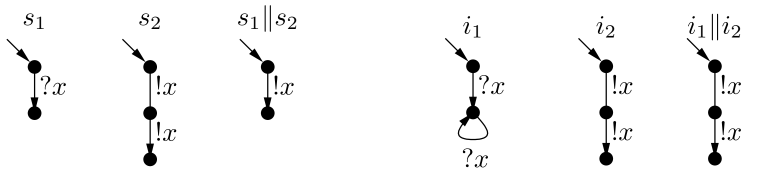


Figure 3.4: Counter-example for parallel composition; see Example 3.3.2

$\delta^* \cup \delta^* \cdot x \cdot \delta^*$ and the suspension traces of s_2 are given by $\{\epsilon, x\} \cup x \cdot x \cdot \delta^*$). We have i_1 **ioco** s_1 and i_2 **ioco** s_2.

After we take the parallel composition of the two specifications we get $s_1\|s_2$, see Figure 3.4 (the corresponding implementation is $i_1\|i_2$). We see that: $out(i_1 \parallel i_2 \textbf{ after } x) = \{x\} \not\subseteq out(s_1\|s_2 \textbf{ after } x) = \{\delta\}$; this means that the parallel composition of the implementations is not **ioco**-correct: $i_1\|i_2$ $\mathbf{io\not{c}o}$ $s_1\|s_2$. □

Analysis shows that i_1 **ioco** s_1, because **ioco** allows underspecification of input actions (a state is underspecified for an action if it does not have any outgoing transitions for the action). However, the semantics of the parallel composition operator does not take underspecification of input actions into account: although s_2 can output a second x, it cannot do so in $s_1\|s_2$, because s_1 cannot input the second x.

It turns out that if we forbid underspecification of input actions, i.e., if the specification explicitly prescribes for any possible input what the allowed responses are, then we do not have this problem. In fact in that case **ioco** is preserved over parallel composition, as we express in the following theorem.

Theorem 3.3.3 Let $s_1, i_1 \in \mathbf{IOTS}(I_1, U_1)$, $s_2, i_2 \in \mathbf{IOTS}(I_2, U_2)$, with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$.

$$i_1 \textbf{ ioco } s_1 \wedge i_2 \textbf{ ioco } s_2 \Rightarrow i_1\|i_2 \textbf{ ioco } s_1\|s_2$$

□

Our running example (Example 3.2.1) shows the same problem illustrated in Example 3.3.2. Although the implementations of the money component and the drink component are **ioco** correct with respect to their specifications, it turns out that the parallel composition of i_{mon} and i_{drk} is not:

$$\begin{aligned} out(i_{\mathsf{mon}}\|i_{\mathsf{drk}} \textbf{ after } \mathsf{i€1} \cdot \mathsf{make\_coffee}) &= \{\mathsf{coffee}, \mathsf{error}\} \\ out(s_{\mathsf{mon}}\|s_{\mathsf{drk}} \textbf{ after } \mathsf{i€1} \cdot \mathsf{make\_coffee}) &= \{\mathsf{coffee}\} \end{aligned}$$

Note that the internal signals are still visible as output actions. To turn them into internal actions is the task of the *hiding* operator, discussed in the next section.

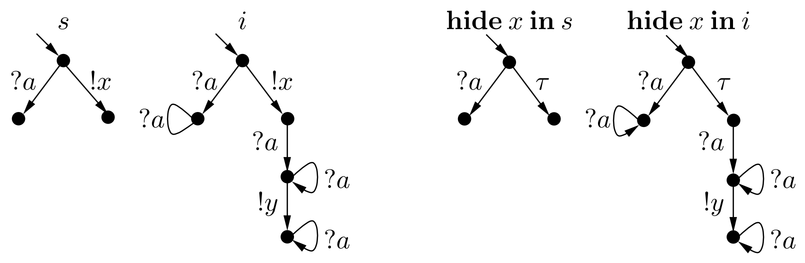


Figure 3.5: Counter-example for hiding

3.3.2 Hiding

In the following example we show that *if* we have a correct implementation according to **ioco**, *then* the implementation does *not* remain correct after hiding (some of the) output actions.

Example 3.3.4 Consider the implementation i and specification s in Figure 3.5, both with input set $\{a\}$ and output set $\{x, y\}$. The specification can perform a choice between inputting an a or outputting an x; after both actions it is underspecified. The implementation can do either one or more a inputs or output an x, input one or more a actions, output y and end with zero or more a inputs. We have i **ioco** s.

When we hide the output action x in the specification and implementation we get specification **hide** $\{x\}$ **in** s, and implementation **hide** $\{x\}$ **in** i (see Figure 3.5 right-hand side). For input a we get: $out($**hide** $\{x\}$ **in** i **after** $a) = \{\delta, y\} \not\subseteq out($**hide** $\{x\}$ **in** s **after** $a) = \{\delta\}$; in other words the **ioco** relation does not hold: (**hide** $\{x\}$ **in** i) **io$\cancel{\text{c}}$o** (**hide** $\{x\}$ **in** s). □

An analysis of the above example shows that s was underspecified, in the sense that it fails to prescribe how an implementation should behave after the trace $!x?a$. The proposed implementation i uses the implementation freedom by having an unspecified y-output after $x{\cdot}a$. When x becomes unobservable due to hiding, the traces $x{\cdot}a$ and a collapse and become indistinguishable: in **hide** $\{x\}$ **in** s and **hide** $\{x\}$ **in** i they both masquerade as the trace a. Now **hide** $\{x\}$ **in** s *appears* to specify that after a, only quiescence (δ) is allowed; however, **hide** $\{x\}$ **in** i still has this unspecified y-output. In other words, hiding creates confusion about what part of the system is underspecified.

Also in this case, if we rule out underspecification, i.e., we limit ourselves to specifications that are IOTSs then this problem disappears. The result is that **ioco** is preserved under hiding, as is stated in the following theorem.

Theorem 3.3.5 If $i, s \in \mathbf{IOTS}(I, U)$ with $V \subseteq U$, then:

$$i \textbf{ ioco } s \;\Rightarrow\; (\textbf{hide } V \textbf{ in } i) \textbf{ ioco } (\textbf{hide } V \textbf{ in } s)$$

□

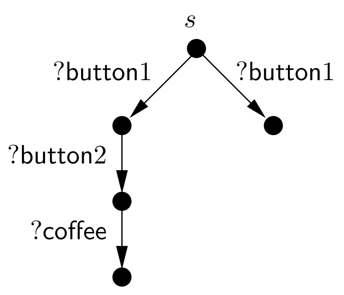


Figure 3.6: Underspecification in **ioco**

3.4 Underspecification

In this section we look into more detail at the **ioco** implementation relation and the reason why **ioco** is not preserved by parallel composition and hiding. We investigate several ways to fix this problem. We start with an analysis of underspecified actions, as this is the root of the **ioco**-preservation problem.

Underspecification comes in two flavors: underspecification of input actions and underspecification of output actions. Underspecification of output actions is always explicit; in an LTS it is represented by a choice between several output actions. The intuition behind this is that we do not know or care which of the output actions is implemented, as long as at least one is. Underspecification of input actions is always implicit; it is represented by absence of the respective input action in the LTS. The intuition behind underspecification of input actions is that after an unspecified input action we do not know or care what the behavior of the specified system is. This means that after an unspecified input action we do not know what actions can be considered correct. From a tester's point of view this means we can stop testing, because everything passes. In other words all behavior after an unspecified input is correct (including quiescence). Following [BHR84] we call this kind of behavior *chaotic*. Note that the interpretation of underspecified actions is to some extent up to the tester. It is an interpretation of the intended behavior of the system under test. Depending on the context there is probably some behavior that the SUT should not show, like for example resetting, halting, exploding, etc. In this text we have the point of view that after an unspecified input action we do not know or care what the behavior is and therefore we deem *everything* correct.

When we look at **ioco** we see that it does not completely adhere to the above notion of underspecified behavior. **ioco** favors specified states over underspecified states. This is easiest explained with an example.

Example 3.4.1 In Figure 3.6 we see a specification of a coffee machine. The idea is to obtain coffee after pressing button1 followed by button2. Because of a non deterministic input of button1 in the start state we can reach two states after button1. One is specified for button2 and the other is not. In the interpretation of underspecification discussed above, the behavior after button1·button2 should be chaotic. However **ioco** looks at the behavior of the states specified for the input action: **ioco** prescribes that $out(s$ **after** button1·button2$) = \{$coffee$\}$. This is behavior that gets us into trouble. □

We have shown in the previous section that **ioco** is not preserved under parallel composition and hiding because of the way it deals with underspecified input actions. Is there a way to fix our **ioco**-preservation problem? One obvious solution is to stick to completely specified specifications: we have shown that **ioco** is preserved under parallel composition and hiding when restricted to IOTSs ($\mathbf{ioco} \subseteq \mathbf{IOTS} \times \mathbf{IOTS}$). We find this solution inadequate because we like the flexibility to work with underspecified specifications. An interesting approach might be to transform LTSs into IOTSs in a way that complies with the notion of underspecification discussed above. There are two other promising candidates. One is to change the semantics of **ioco** in such a way that it complies with our notion of underspecification. The other is to change the semantics of the parallel composition and hiding operators. In the remainder of this section we investigate the following approaches to preserve **ioco** over parallel composition and hiding.

- A transformation of LTSs to IOTSs in such a way that it complies with our notion of underspecification. Theorem 3.3.3 gives us the wanted preservation result.
- A change of the **ioco** definition in such a way that it does not favor specified behavior over underspecified behavior.
- A new definition of the parallel composition operator in such a way that underspecified input actions are taken into account and treated properly, compliant with our notion of underspecified behavior.

3.4.1 Completion

A way to transform an LTS to an IOTS that is often found in the literature is *angelic* completion [DNS95]. This form of completion adds to states that are underspecified for a certain input action, a self-loop (a transition ending in the same state) labeled with the lacking input action. When we do this for all input actions we end up with an input enabled transition system. The problem with this kind of completion is that it does not capture the notion of implementation freedom that we discussed above, namely that after an underspecified input action we do not know or care what the behavior of the specified system is. Angelic completion expresses that we ignore unspecified input actions and otherwise behave as specified. This does not help us in our search to find an implementation relation that works directly on LTSs as we illustrate in the following example.

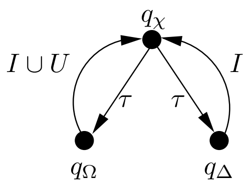


Figure 3.7: Demonic completion process

Example 3.4.2 Let us look again at the counter-example for hiding in Figure 3.5 on page 52. The specification s prescribes that after the output action x the input action a is underspecified. Therefore we would like to see that we do not know or care what the behavior of the system is after $x{\cdot}a$. To put it formally we want $out(s\ \mathbf{after}\ x{\cdot}a) = U_\delta$. Angelic completion adds a self loop with a in the state reached after x. This means that $out(s\ \mathbf{after}\ x{\cdot}a) = \{\delta\}$. This does not capture the notion of implementation freedom that we are looking for. □

In translating LTSs to IOTSs, we propose to model the implementation freedom after an unspecified input action via a chaotic process shown in Figure 3.7. We model chaotic behavior through a state q_χ with the property: $\forall \lambda \in U : q_\chi \xRightarrow{\lambda} q_\chi$ and $\forall \lambda \in I : q_\chi \xRightarrow{\delta^*\cdot\lambda} q_\chi$ (where χ stands for chaos). Secondly, we add for every stable state q (of a given LTS) that is underspecified for input a, a transition (q, a, q_χ). This turns the LTS into an IOTS. After [DNS95] we call this procedure *demonic* completion as opposed to *angelic* completion. There is a practical choice to make in the operationalization of what *chaotic* means. In this approach we choose to define chaotic as all *possible* behavior relative to the label set of the transition system. Another choice might be to allow all possible actions relative to a global label set. Our choice is a pragmatic one, because it is easy to express with regular transition systems. The problem with a global label set is that we need to know which actions are input actions. A priori this is not clear and depends on the specific transition system. A good solution for this would be to use Heerink's multi input output transition systems [Hee98]. This type of transition system identifies channels over which it can communicate with the environment. This makes it possible to differentiate between input channels and output channels and thus makes it possible to point out input actions from a global label set.

Definition 3.4.3 gives the definition of demonic completion. The state set Q is extended to Q' by adding the demonic states q_χ, q_Ω and q_Δ. The set of transitions T is extended to T' as discussed above: all stable states that are underspecified for an input action in the label set of the machine get a transition with the underspecified input action to q_χ.

Definition 3.4.3 [demonic completion] $\Xi : \mathbf{LTS}(I, U) \to \mathbf{IOTS}(I, U)$ is defined by $\langle Q, I, U, T, q_0\rangle \mapsto \langle Q', I, U, T', q_0\rangle$, where

$$\begin{aligned}
Q' = Q \quad & \cup\{q_\chi, q_\Omega, q_\Delta\}, \text{ where } q_\chi, q_\Omega, q_\Delta \notin Q \\
T' = T \quad & \cup\{(q, a, q_\chi) \mid q \in Q, a \in I, q \stackrel{a}{\not\longrightarrow}, q \stackrel{\tau}{\not\longrightarrow}\} \cup \{(q_\Delta, \lambda, q_\chi) \mid \lambda \in I\} \\
& \cup\{(q_\chi, \tau, q_\Omega), (q_\chi, \tau, q_\Delta)\} \cup \{(q_\Omega, \lambda, q_\chi) \mid \lambda \in L\}
\end{aligned}$$

Example 3.4.4 In Figure 3.8 we show the demonically completed specification from the hiding counter-example of Figure 3.5 on page 52. The two final states are underspecified for action a and therefore we added for both states a transition labeled with a to the chaotic state q_χ. The input action a after action x is underspecified, therefore all behavior is allowed after a (chaos). At the right-hand side we show the demonically completed specification ($\Xi(s)$) after hiding action x. We have $out(\textbf{hide}\, x\, \textbf{in}\, \Xi(s)\ \textbf{after}\ a) = U_\delta \backslash \{x\}$. This is exactly the notion of underspecification that we are looking for. □

An important property of demonic completion is that it only adds transitions from *stable* states with underspecified inputs in the original LTS to q_χ. Moreover, as expressed in Proposition 3.4.5, it does not delete states nor transitions. The chaotic IOTS acts as a sink state: once one of the added states (q_χ, q_Ω or q_Δ) has been reached, the chaotic IOTS will never be left.

Proposition 3.4.5 Let $s \in \mathbf{LTS}(I, U)$.

$$\forall \sigma \in L_\delta^*, q' \in Q_s : s \stackrel{\sigma}{\Longrightarrow} q' \Leftrightarrow \Xi(s) \stackrel{\sigma}{\Longrightarrow} q'$$

□

We use the notation "**ioco** $\circ\, \Xi$" (**ioco** after Ξ) to denote that before applying **ioco**, the LTS specification is transformed to an IOTS by Ξ; i.e., to put it formally i (**ioco** $\circ\, \Xi$) $s \Leftrightarrow i$ **ioco** $\Xi(s)$. Theorem 3.4.6 expresses that this relation is weaker than **ioco**. Note that despite the terminology, it is a good thing that **ioco** $\circ\, \Xi$ is weaker, because **ioco** was too restrictive (in

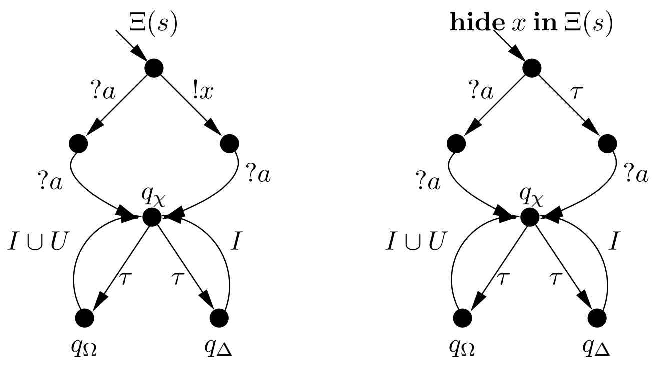


Figure 3.8: Demonic completion in combination with hiding

the sense that it prefers specified behavior over underspecified behavior). Weaker in this case means that previously conformant implementations are still conformant, but it might be that previously non-conformant implementations are now allowed with this new notion of conformance. This is easy to understand. We do not change the already defined behavior of the transition system, therefore we will never make the set of allowed outputs after a certain trace smaller. However, it might be the case that we make the set of allowed outputs bigger after a certain trace, if this trace contains underspecified behavior. When we look for example at Figure 3.6 on page 53 we see that the output of tea after button1·button2 is not allowed. However after demonic completion it is allowed, because button2 is unspecified after button1 (in the right branch).

Theorem 3.4.6

$$\textbf{ioco} \subseteq \textbf{ioco} \circ \Xi$$

□

Note that the opposite of Theorem 3.4.6 is not true i.e., $i\ (\textbf{ioco} \circ \Xi)\ s \not\Rightarrow i\ \textbf{ioco}\ s$ (as Example 3.4.4 and the counter-examples for parallel composition and hiding of Section 3.3 show).

The results in this section make it possible to test an integrated system by comparing the individual components to their *demonically completed* specifications. If the components conform, then the composition of implementations also conforms to the composition of the demonically completed specifications. The same holds for hiding.

Corollary 3.4.7 Let $s_k \in \textbf{IOLTS}(I_k, U_k)$ and $i_k \in \textbf{IOTS}(I_k, U_k)$ for $k = 1, 2$, $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$, $V \subseteq U$,

$$i_1\ \textbf{ioco}\ \Xi(s_1) \wedge i_2\ \textbf{ioco}\ \Xi(s_2) \Rightarrow i_1 \| i_2\ \textbf{ioco}\ \Xi(s_1) \| \Xi(s_2)$$
$$i_1\ \textbf{ioco}\ \Xi(s_1) \Rightarrow \textbf{hide}\ V\ \textbf{in}\ i_1\ \textbf{ioco}\ \textbf{hide}\ V\ \textbf{in}\ \Xi(s_1)$$

3.4.2 From ioco to uioco

In this section we define the implementation relation **uioco** that has the characteristics of **ioco** ∘ Ξ, but works directly on LTS specifications. **uioco** uses a subset of the *Straces*, which we call *Utraces*, which stands for *Universal* traces.

Definition 3.4.8 [*Utraces*] Let $s \in \textbf{LTS}(I, U)$.

$$\begin{aligned} \mathit{Utraces}(s) =_{\mathrm{def}} \{\sigma \in \mathit{Straces}(s) \mid \nexists q \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, a \in I : \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2 \\ \wedge\, s \xRightarrow{\sigma_1} q \wedge q \overset{a}{\not\Rightarrow} \} \end{aligned}$$

Intuitively, the *Utraces* are the *Straces* without the traces with underspecified input actions; so-called underspecified traces. To put it in terms of the definition above, a trace σ is underspecified if there exists a prefix $\sigma_1 \cdot a$ of σ which can bring s via trace σ_1 in a state q that is underspecified for a ($s \overset{\sigma_1}{\Longrightarrow} q$ and $q \overset{a}{\not\Longrightarrow}$).

When we use *Utraces* instead of *Straces* we obtain a variant of **ioco** that we call **uioco**.

Definition 3.4.9 [**uioco**] Let $s \in \mathbf{LTS}(I, U), i \in \mathbf{IOTS}(I, U)$

$$i \textbf{ uioco } s =_{\text{def}} \forall \sigma \in \mathit{Utraces}(s) : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(s \textbf{ after } \sigma)$$

In the following theorem we state that **uioco** is equivalent to $\mathbf{ioco} \circ \Xi$. This equivalence is quite intuitive. $\mathbf{ioco} \circ \Xi$ uses extra states to handle underspecified behavior, which are constructed to display chaotic behavior. If $\Xi(s)$ reaches such a state, then all behavior is considered correct. **uioco**, on the other hand, circumvents underspecified behavior, because it uses *Utraces*.

Theorem 3.4.10

$$\mathbf{uioco} = \mathbf{ioco} \circ \Xi$$

□

It is important to note that **uioco** is not compositional, because of the same problems as **ioco** (see Example 3.3.2; underspecified input actions are blocking). The benefit of **uioco** lies in the fact that we can test the components with the same power as $\mathbf{ioco} \circ \Xi$ without the extra effort of demonic completion.

3.4.3 Changed semantics for the parallel operator

The last option we investigate is to change the semantics of the parallel composition. The approach is reminiscent of demonic completion in that we introduce transitions to a chaotic process. In Table 3.1 we show the SOS rules for the new parallel composition operator. We denote the new parallel composition operator with][.

We introduce three processes χ, Δ and Ω (demonic process). χ goes nondeterministically via an internal transition to Δ or Ω. Ω can do all possible actions from L and Δ is a quiescent state that can only do input actions. The label sets I and L are relative to the process in question.

The parallel composition operator looks quite similar to the original definition from Table 2.2. The main difference is when one of the processes in the composition cannot do its input action; the situation where the parallel composition would be blocked. The new situation is that the process

Operator	Transition rules
Demonic process	$\chi \xrightarrow{\tau} \Delta \quad \chi \xrightarrow{\tau} \Omega$ $\forall \lambda \in I : \Delta \xrightarrow{\lambda} \chi \quad \forall \lambda \in L : \Omega \xrightarrow{\lambda} \chi$
$B_1][B_2$	1) $\dfrac{B_1 \xrightarrow{\mu} B_1', \mu \in (L_{B_1} \backslash L_{B_2}) \cup \{\tau\}}{B_1][B_2 \xrightarrow{\mu} B_1'][B_2}$ 2) $\dfrac{B_2 \xrightarrow{\mu} B_2', \mu \in (L_{B_2} \backslash L_{B_1}) \cup \{\tau\}}{B_1][B_2 \xrightarrow{\mu} B_1][B_2'}$ 3) $\dfrac{B_1 \xrightarrow{\mu} B_1', B_2 \not\xrightarrow{\mu,\tau}, \mu \in U_{B_1} \cap I_{B_2}}{B_1][B_2 \xrightarrow{\mu} B_1'][\chi}$ 4) $\dfrac{B_1 \not\xrightarrow{\mu,\tau}, B_2 \xrightarrow{\mu} B_2', \mu \in U_{B_2} \cap I_{B_1}}{B_1][B_2 \xrightarrow{\mu} \chi][B_2'}$ 5) $\dfrac{B_1 \xrightarrow{\mu} B_1', B_2 \xrightarrow{\mu} B_2', \mu \in L_{B_1} \cap L_{B_2}}{B_1][B_2 \xrightarrow{\mu} B_1'][B_2'}$

Table 3.1: SOS rules for the new parallel composition operator

blocking the composition makes a transition for the unspecified input action to the demonic process χ. We leave out the set of actions G over which the processes synchronize, because we always synchronize over the intersection of the label sets, under the condition that for two systems r and s the following restriction holds over their label sets: $I_r \cap I_s = U_r \cap U_s = \emptyset$. When we compose two systems r and s the signature of the composition is $U_{r][s} = U_r \cup U_s$ and $I_{r][s} = I_r \backslash U_s \cup I_s \backslash U_r$. The new definition preserves **ioco**, as we show in the following theorem.

Theorem 3.4.11 Let $s_k \in \mathbf{IOLTS}(I_k, U_k)$ and $i_k \in \mathbf{IOTS}(I_k, U_k)$ for $k = 1, 2$ with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$.

$$i_1 \ \mathbf{ioco} \ s_1 \wedge i_2 \ \mathbf{ioco} \ s_2 \Rightarrow i_1][i_2 \ \mathbf{ioco} \ s_1][s_2$$

□

This result does not come as a surprise, because it is very similar to our completion result. We disable blocking of input actions by jumping to a chaotic process (via the underspecified input action).

We can define new semantics for the hiding operator in a similar fashion, i.e., by changing the hiding rules when we encounter a state that is underspecified for an input action. We can also test hiding with the **uioco** relation. For **uioco** circumvents underspecified behavior that the new hiding definition would lead to the demonic process.

The results of this section make it possible to do compositional testing with **ioco** without first making the model input complete. We think that this lowers the barrier to use compositional testing with **ioco** in practice. Splitting functionality in components (divide and conquer) is a powerful engineering technique that can now be used for model-based testing.

3.4.4 Chaos and convergence.

While treating our approach to cope with underspecification of input actions we swept something under the rug. The **ioco** theory demands that the LTS specifications are strongly convergent. In short this means that infinite τ-loops are forbidden. With the introduction of the chaotic process in completion and the improved semantics we run the risk of introducing a non-convergent system when we apply hiding. This happens when we hide one of the output actions in the chaotic process: we introduce a τ-transition from q_Ω to q_χ, combined with the τ-transition from q_χ to q_Ω we have a τ-loop.

There are several ways to treat this issue. The most thorough solution is to lift the convergence restriction from **ioco**. For example, we could interpret infinite τ-loops as δ transitions, the way that [JJ05] do. We find this approach is very interesting for further research. A more pragmatic approach is to forbid τ-transitions from q_Ω (and q_Δ), or to restrict the output set of the chaos process to the output set of the system after hiding. This is not a nice solution, because it means that the chaos process is treated differently from a normal process, or a normal transition system. Another take on the issue is to stop testing when the specification hits chaotic behavior. Chaos means underspecified behavior and one can argue that before continuing testing we should improve the specification. It is already very helpful that we can identify this type of underspecification. This means that once we hit the chaotic state q_χ we stop testing and as a result we do not encounter possibly introduced τ-loops. We investigate this line of thought in the following paragraph.

Chaos and testing. Knowledge of underspecification can be used for test generation and test execution. We can stop testing after an underspecified input, because every implementation will always pass (the chaotic process allows all possible actions). Automatic test generation will yield many spurious tests if we do not keep track of underspecified behavior, for example with special states or with special labels that signal chaotic behavior. With demonic completion and our adapted parallel operator we can keep track of underspecified behavior via the demonic process. We could for example extend the test generation algorithm from Definition 2.5.5 with a rule that whenever the state q_χ (or χ) is in the set of states S we stop testing via a **pass** verdict. On the other hand ending signaling that we have hit underspecified behavior seems a good alternative, because this signals that an undesirable situation has happened. In our experience, hitting underspecified behavior while testing is in general undesired. Another option would be to change the test generation algorithm in such a way that it reflects the **uioco** relation, avoiding underspecification altogether.

An interesting situation occurs when testing two (or more) processes in parallel and one of the processes becomes chaotic. What does this mean for testing? Can we continue testing or should we stop? Technically we can continue testing, since part of the composition is not chaotic. For this part we can test its specified behavior. However from a practical point we argue that it is better to stop testing and to update the model. Given that it is practically possible, we think it is better to test specified behavior. If the model hits underspecified behavior this might indicate that the model is not good enough and needs fixing.

3.5 Testing in context

We have discussed the **ioco**-preservation properties mainly in the context of compositional testing, but the results can easily be transposed to testing in context. With testing in context we mean the following. Quite often it is the case that the IUT that we want to test cannot be tested directly, but only indirectly via some kind of interface. With *context* we refer to the software/hardware needed to access the IUT.

Example 3.5.1 Suppose that we want to test the drink preparing component of our example in Figure 3.2 on page 48 (left-hand side) and that we can only test the whole machine, i.e., we have to test the drink component via the money component (right-hand side). In that case we say that the money component is the test context via which we access the drink preparing component. In order to be able to test the drink preparing component we can take the parallel composition of the money and drink preparing components with hiding the commands between the components if necessary. This is exactly what we have been investigating in the previous sections. □

Suppose an implementation under test i is tested via a context c. The tester interacts with c, and c interacts with i; the tester cannot directly or only partially interact with i. Then we have $I_i \subseteq U_c$ and $U_i \subseteq I_c$, and L_i is not observable for the tester, i.e., hidden. The tester observes the system as an implementation in a context in the following way: $\mathcal{C}[i] = \textbf{hide}\ (I_i \cap U_c) \cup (I_c \cap U_i)\ \textbf{in}\ c \| i$. Now Theorem 3.3.3 and Theorem 3.3.5 directly lead to the following corollary for testing in context.

Corollary 3.5.2 Let $s, i \in \textbf{IOTS}$ occur in an error-free test context $\mathcal{C}[\_]$.

$$\mathcal{C}[i]\ \textbf{io}\not\textbf{co}\ \mathcal{C}[s] \Rightarrow i\ \textbf{io}\not\textbf{co}\ s$$

Hence, an error detected while testing the implementation in its context is a real error of the implementation, but not the other way around: an error in the implementation may not be detectable when tested in a context. This holds of course under the assumption that the test context is error free, otherwise an error found in the combination test context and implementation can also be the result of an error in the test context.

3.6 Conclusions

The results of this chapter show that standard **ioco** cannot be used for compositional testing. The main problem is the underspecification of input actions. When specifications are modeled as IOTSs, compositional testing with **ioco** works fine; see Theorem 3.3.3 and Theorem 3.3.5.

The insights gained from these results can be explained in terms of *underspecification*. **ioco** recognizes two kinds of underspecification: omitting input actions from a state (which implies a *don't care* if an input does occur) and including multiple output actions from a state (which allows the implementation to choose between them). It turns out that the first of these two is incompatible with parallel composition and hiding when using LTSs as models.

We discussed three approaches in our search for a solution for compositional testing: completion of LTS specifications, **uioco** and the new parallel operator '][$'$. For completion we introduced the function Ξ to demonically complete an LTS specification, i.e., transform an LTS to an IOTS in a way that captures our notion of underspecification. This means that the above results become applicable and the **ioco** theory with completed specifications can be used for compositional testing. The resulting relation is slightly weaker than the original **ioco** relation; previously conformant implementations are still conformant, but it might be that previously non-conformant implementations are allowed under the modified notion of conformance. This is because **ioco** favors specified behavior over unspecified behavior. In that sense **ioco** was too restrictive.

Testing after completion is in principle (much) more expensive since, due to the nature of IOTSs, even the completion of a finite specification already displays infinite testable behavior. We argued that for practical testing it is necessary to keep track of chaotic behavior. This reduces the number of tests and the length of tests dramatically because we can stop testing when the specification becomes chaotic. An implementation relation that partially remedies this problem is **uioco**. This relation enables us to use the original component specifications, *before* completion, for compositional testing (see Theorem 3.4.10). However, **uioco** is not compositional. It runs into the same problems as **ioco** because the specifications are allowed to be LTSs.

Our final approach to preserve **ioco** is to change the semantics of the parallel operator to reflect the desired notion of underspecification. We showed that **ioco** is preserved for this new operator, meaning that we can use it for compositional testing. Like in the completion case, with practical testing, we need to keep track of the introduced chaotic behavior.

What is the practical implication of the results of this chapter? What is clear is that parallel composition and hiding cannot be used for the **ioco** theory, because their semantics is inappropriate for testing. The new results can be used as long as we stop testing when we hit underspecified behavior. It is not entirely clear what testing with chaotic components means. On the one hand we have the theoretical problem that we might introduce divergent behavior. On the other hand we have the practical problem that it is unclear if testing with chaotic components is worthwhile. We think that the results show that underspecified behavior is an important concept for the **ioco** theory and for testing in general and deserves a more prominent place. It shows that concepts that are commonly used, like parallel composition, cannot a priori be used for testing, when implementations are considered to be input-enabled and specifications are not. On a more general level, with the results of this section in hand, it might not be a bad idea to re-evaluate the semantics of process operations, like the ones of Section 2.3.2. Is blocking a realistic property of the parallel operator? We think that the new parallel operator that we introduced is a better alternative.

Traditionally conformance testing is seen as the activity of checking the conformance of a single black box implementation against its specification. The testing of communicating components is often considered to be outside the scope of conformance testing. The results of this chapter show that testing communicating components is not outside the realm of conformance testing anymore.

Acknowledgments. We want to thank D. Lee and A. Petrenko for sharing their knowledge of FSM testing and for their insightful discussions.

Chapter 4

Action Refinement and Model-Based Testing

In this chapter we provide the rationale for our research in action refinement in model-based testing. We start with an introduction into the problem and we treat several action refinement examples that we collected during our research. Next we introduce the action refinement ingredients that we identify for model-based testing and define the research questions that we want to answer. We continue with an overview of the research found in the literature on action refinement and we discuss in how far existing theory can help us in our research. We introduce an action refinement classification, based on the action refinement scenarios and the examined existing action refinement theory. We conclude with a discussion of the type of action refinement we have chosen to investigate and that we will present in the next chapter.

4.1 Introduction

The model in model-based testing plays a crucial part. If the model is not a faithful representation of the behavior of the IUT we may derive useless test-cases from it. We may get test-cases that give a **pass** verdict where they should give a **fail** verdict, or the other way around, test-cases that give a **fail** verdict where they should give a **pass** verdict (unsound test-cases). Another possibility are test-cases that are not executable because they are incompatible with the IUT, for example when they do not have the right commands. What to do when we are in the situation that the model is not a faithful representation of the IUT? When we posses the model from which we derived our test-set, we can correct it and regenerate the test-set. But when we do not have the model, we can only change the test-cases. We could of course change the model and/or test-cases ad hoc by hand, but we find this error-prone and time consuming. The problem that we

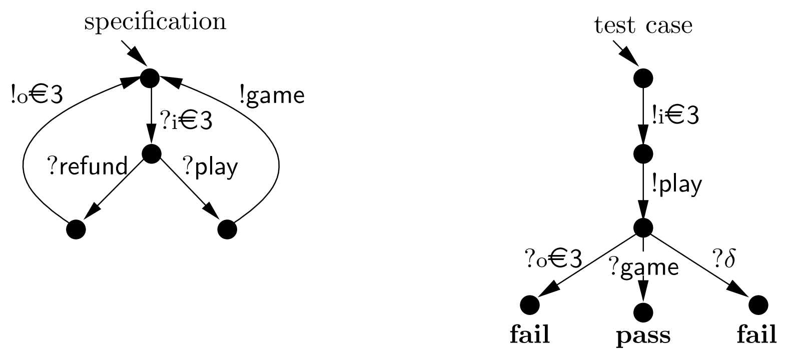


Figure 4.1: Video game example

investigate in this chapter is whether we can change the models and test-cases in a controlled and automatic manner. We use the term *refinement* to refer to the changing of a model or test-case. We introduce refinement with an example.

Example 4.1.1 In Figure 4.1 we show an example of a specification (left-hand side) and a test-case (right-hand side) of a video game. The specification tells us that we can enter three euro, after which we can press the refund-button to get our money back, or press the play-button to play the video game (we use a subscript 'i' or 'o' to indicate whether the action is in the input or output label set, as these label sets need to be disjoint). The test-case tests the system by inputting three euro and pressing the play-button, after which the response of the system is observed: only game is considered to be a correct response.

Suppose that we want to refine the three euro input and output actions into one euro followed by two euro or vice versa (we abstract from all the other possibilities to compose three euro). Figure 4.2 on the next page gives an example of a system specification and test-case that we want to obtain via refinement. In the refined specification we see that we can either input one euro followed by two euro, or vice versa, after which we can press the play-button and play the video game, or press the refund-button and get our money back (two euro followed by one euro, or vice versa). It is also possible to get a refund after we have inserted one, or two euro. On the right-hand side of the figure we see an example of a refined test-case. We enter one euro, followed by two euro, after which we press the play-button and observe the response of the system: only the observation of game leads to pass. □

The central question is how to obtain refined test-cases. In Chapter 5 we present our approach and results in answering this question. This chapter is a preliminary study of the issues that arise when studying action refinement

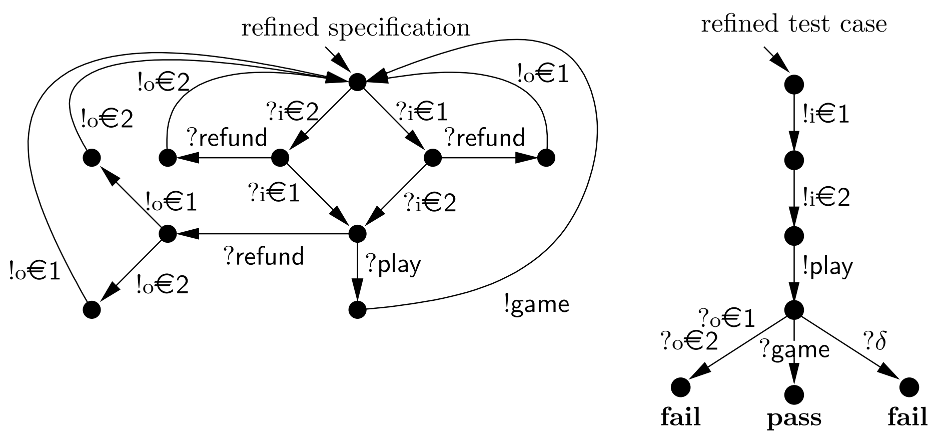


Figure 4.2: Refined video game example

in model-based testing. To set the scene, we start with some refinement scenarios that we collected during our research in Section 4.2. After that we present the ingredients in our approach to action refinement in model-based testing in Section 3.2. We give an overview of the existing action refinement research and discuss to what extent we can use it for model-based testing in Section 4.4. We end with a discussion of the specific type of action refinement that we chose to investigate in Section 4.6.

4.2 Action refinement scenarios

During our research we collected several examples of action refinement in model-based testing. Some examples were developed by ourselves, others come from work with business partners. We present these examples, or scenarios as we call them, to give an introduction by example and to illustrate the type of problems that we want to solve with our research. We start with quickly introducing some basic refinement concepts that we use in the scenarios.

4.2.1 Refinement function

In this thesis we restrict ourselves to a well-studied refinement approach, called action refinement [GR01]. Action refinement replaces an action in the model or test-case by more refined behavior. The idea behind action refinement is top-down systems design: start with an abstract design that captures the most important parts of the system and gradually add more detail. Action refinement is a way to add more detail to a specification in a formal and controlled manner. This is exactly what we want to do in our model-based testing situation. In order to obtain a specification with the required level of detail, we refine the abstract system specification using action refinement. Likewise, we want to obtain refined test-cases, using action refinement.

In the action refinement scenarios we use a so called *refinement function*. This function relates abstract actions to labeled transition systems. With the refinement function we can record action refinement, the refined system and the refinement transition system. We use a Final State LTS for the refinement function: an LTS with an explicit end state.

Definition 4.2.1 [Final State Labeled Transition System] A Final State Labeled Transition System (FLTS) is a six tuple $\langle Q, I, U, T, \mathsf{start}, \mathsf{final}\rangle$ such that $\langle Q, I, U, T, \mathsf{start}\rangle$ is an LTS. $\mathsf{final} \in Q$ denotes the end state, with $\mathsf{start} \neq \mathsf{final}$.

We denote the class of all FLTSs over I and U by $\mathbf{FLTS}(I, U)$. We denote the refinement function as: $r : L \to \mathbf{FLTS}(I, U)$. When the label sets are obvious we may leave them out in the refinement function. We use the term *refinement transition system* to refer to the refinement behavior expressed in an FLTS. We use the notation $r(\lambda)$ to denote the refinement transition system for abstract action λ. Thanks to the final state we know when the refinement behavior ends successfully. We denote the refinement of a transition system s (for example a specification or a test-case) by $s[r]$. We show how this notation is put to use in the following example.

Example 4.2.2 In Figure 4.3 we give some refinement transition systems for our video game example from the introduction (see Example 4.1.1). With these refinement transition systems we express what the concrete behavior of the abstract action we refine should be. We see that the abstract output action o€3 is refined by the choice between o€1 followed by o€2 or in the inverse order. The final state is denoted by a circle around the state. The refinement transition system of the input action i€3 is slightly more complex, because we allow a refund of the thus far entered money. The refinements of game, refund and play are straightforward (we do not change the behavior of the abstract action). We only show the refinement transition system for game. With these refinement transition systems we want to refine the abstract video game system and test-cases as shown in Figure 4.1 on page 66. □

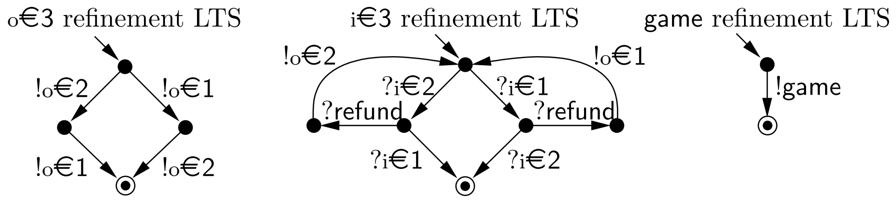


Figure 4.3: Refinement transition systems for our video game example

4.2.2 Linear output splitting

The first scenario comes from a car navigation system manufacturer in the Netherlands. In the example we treat the part that guides a driver to take the next exit on the high way. A command in the abstract specification consists of several commands in the implementation. We call it linear output splitting, because as you will see, there is no branching behavior in the refinement and only output actions are involved. We show the abstract system and abstract test-cases and discuss the refined specification and test-cases that we want to obtain.

Example 4.2.3 On the left-hand side in Figure 4.4 on the following page we see part of a specification that guides a driver to take the next exit on the highway in a number of meters, in this example 200 meters. In the specification this consists of four sound parts (the parts are split up in this way to enable reuse). The sound parts are 1) 'Take the next exit', 2) 'in' 3) '200' 4) 'meters'. The implementation is more refined in that it also tells the driver the number of the exit. In the middle of Figure 4.4 on the next page we show the refined specification that we want to obtain. The refined specification consists of six steps: 1) 'Take the next exit number' 2) the number of the exit, we use '15' for this example 3) 'take the next exit' 4) 'in' 5) '200' 6) 'meters'. We express this concrete behavior as a refinement of 'take the next exit' as shown in the refinement transition system on the right-hand side of the figure.

As some background information, to interpret the example, in the implementation of the navigation system, the text is spoken using a specific timing, based on the gps location of the car. This means that first the up-coming exit is announced with: 'Take the next exit, number 15'. When the car is within 200 meters of the exit, the driver is advised to take the exit with: 'Take the next exit in 200 meters'. We abstract from the timing information in our example. On the left-hand side of Figure 4.5 on page 71 we show part of an abstract test-case derived from the abstract specification. We abstract from the test steps occurring before the observation "take the next exit", denoted by the dashed arrow, and we abstract from the test steps occurring after the observation, denoted by the triangle. With the term *otherwise* we abbreviate all other possible observations. An example of a refined test-case that we want to obtain is shown on the right-hand side of the figure. The behavior of the abstract action take the next exit is split in three observation steps. □

4.2.3 Calculator

In this section we give an example of action refinement involved in testing a calculator (such as a calculator found on a computer or a hand calculator).

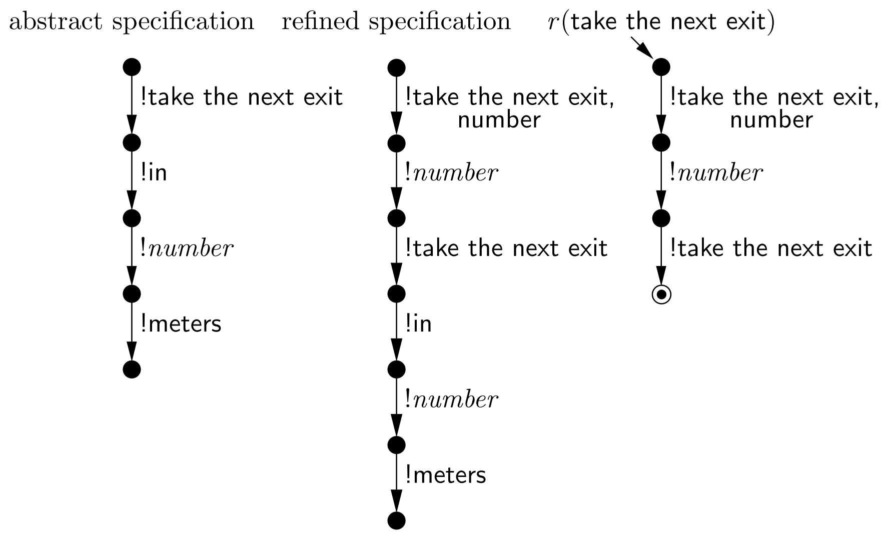


Figure 4.4: Specification, implementation, refinement transition system for navigation example

Where the specification talks about inputting *numbers*, in the implementation numbers are entered digit by digit.

Example 4.2.4 In Figure 4.6 on the next page we give the specification of the calculator as an LTS (left-hand side). In the start state (q_0) we can enter a number by which we make the transition to state q_1 (we write inumber for inputting a number and likewise onumber for outputting a number). In q_1 we can either input an operation (op) or choose to obtain a result by pressing the '=' key. If we input an operation we go to state q_2, in this state we can again input a number and go back to state q_1. When we input '=' in state q_1 we make the transition to state q_3 where the system will output the result as a number.

The implementation has some more detail than the specification: numbers are entered digit by digit and there are four types of operations (+, $-$, $\times$ and /). On the right-hand side of Figure 4.6 on the facing page we show a specification with the required level of detail to generate test-cases from.

In Figure 4.7 on page 72 we show transition systems that define the refined behavior of inumber and the operation op. In the refinement of inumber (left-hand side), we enter numbers in our calculator by pressing single digits. So instead of the single action inumber, entering a number consists of at least one digit followed by zero or more other digits. In the refinement of the operation (right-hand side), we can choose between 4 operations: $+, -, \times, /$.

With this refinement information we want to refine the abstract test-case on the left-hand side of Figure 4.8 on page 73 to the refined ones, for example the one shown on the right-hand side. □

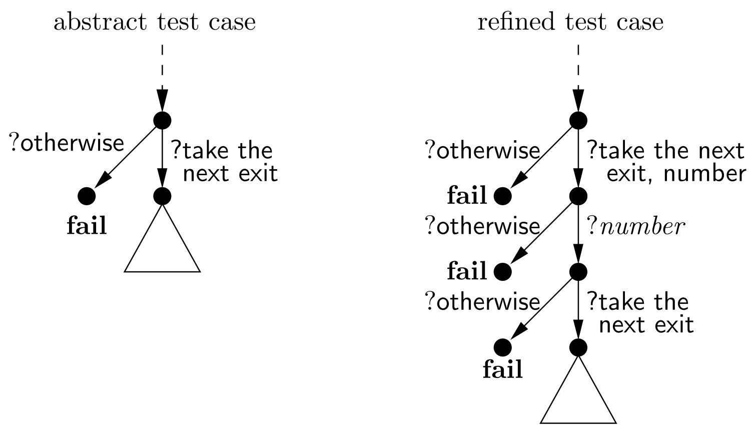


Figure 4.5: Abstract and refined test-case for the navigation example

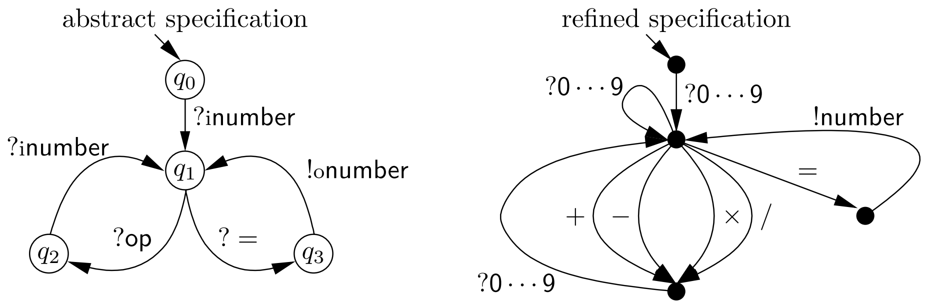


Figure 4.6: Abstract and refined calculator specification

Notice that this rather simple example can easily be extended to a more realistic calculator. Think of extending it with the possibility to correct entries of numbers or operations, real calculations, more and more complex operations, negative numbers, decimal fractions, etc. This also increases the complexity of our refinements.

4.2.4 Remote procedure call

Remote procedure calls (RPC) are often used in computer programs. With a remote procedure call, one can execute a procedure on a remote component. In general, the implementation of an RPC first sends a request to the remote party after which the remote party sends the answer to the requester. Specifications tend to abstract from this 2-phase process. Let us illustrate this with an example from an embedded system that controls a laser.

Example 4.2.5 In this example we use a (remote) procedure that returns the status of the laser. We focus mainly on the refinement transition system and abstract from the refined specification and test-cases (these are analogous to the other examples). The specification uses the action LaserStatus to model the RPC. In the implementation it turns out to be implemented by two actions: callLaserStatus followed by getLaserStatus. This means that we cannot execute test-cases with the abstract action LaserStatus. In order to obtain refined models and/or test-cases we define the refined behavior by the refinement transition system in the middle of Figure 4.9 on the next page. With the refinement transition system we want to refine the abstract system on the left-hand side and obtain refined test-cases.

Note that this refinement is not complete in that it only models the positive behavior. What happens if we do not get an answer from our remote party? This is an interesting scenario. If we want to relate the absence of the answer to already specified behavior we need a way to express this. We have not yet seen a solution for this. In case we do not connect it to already specified behavior, we can express it as follows. On the right-hand side of Figure 4.9 on the facing page we show an example of a refinement transition system that takes other input into account. The triangle in the transition system denotes a continuation of the transition system (from which we abstracted). □

An interesting observation with remote procedure calls is that the abstract actions may not have direction (they are neither input nor output actions, or one might say they are both), whereas in the refined case they do have direction. This seems typical for system design that starts with an abstract design and gradually adds information. Also some of the other scenarios can be presented this way. For example the initialization of a laser component in Example 4.2.7 can be seen as one abstract action without direction: initialize laser. Likewise Example 4.2.8 features a direction-less login action in the abstract specification. As far as we know this kind of refinement has not been studied.

Another scenario encountered in practice, similar to the previous example, is that in order to execute an RPC in the implementation we have to perform some other actions. We show this behavior in the following example in which we switch a laser on via the LaserOn command.

Example 4.2.6 The specification, left-hand side in Figure 4.10 on page 74, prescribes that we switch a laser on by issuing the LaserOn command. In

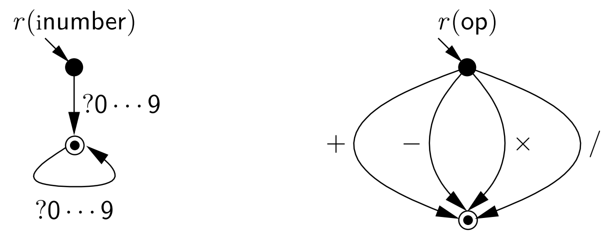


Figure 4.7: Calculator refinement transition systems

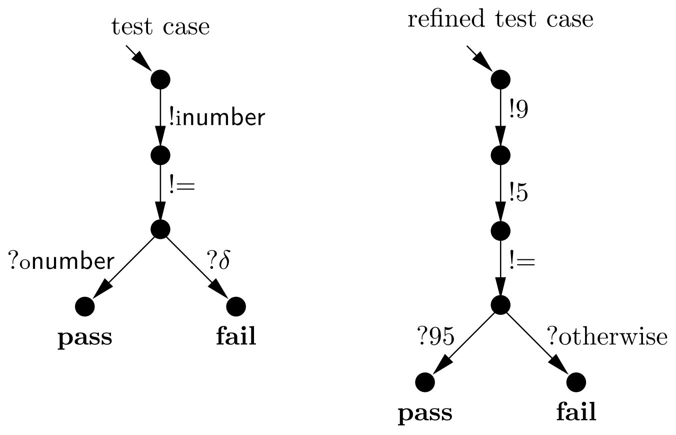


Figure 4.8: Abstract and refined calculator test-cases

practice, the LaserOn command requires 2 parameters: a and b, that we have to query first. In this example we query the parameter values sequentially. We first issue requestA to signal another part of the system to give a value for parameter a. We get the value via the returnA command. After we obtain the values we can issue the switchLaserOn command with values a and b. The refinement transition system is shown on the right-hand side of Figure 4.10 on the following page.

□

For this kind of refinement it is important to know the dependencies between requestA and requestB. If they are allowed to be executed in arbitrary order we need more complex behavior than the one shown in Figure 4.10 on the next page, as this shows the refined behavior in the case that requestB is executed *after* requestA. What to do if requestB is also allowed to occur *before* requestA? We analyze this problem in some more detail, as well as the impact on test-cases, in the following scenario.

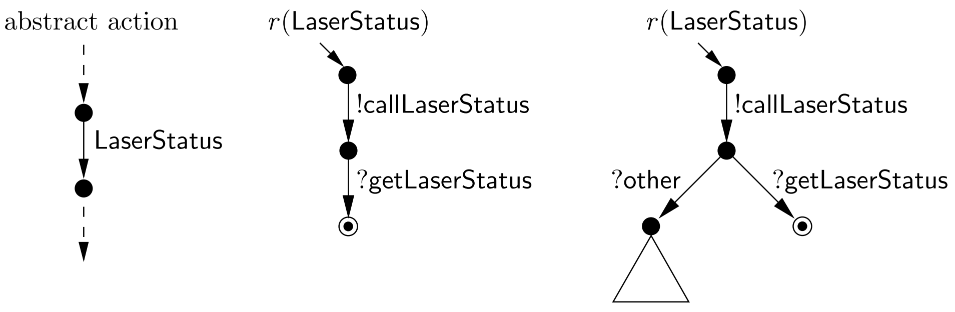


Figure 4.9: Remote procedure call

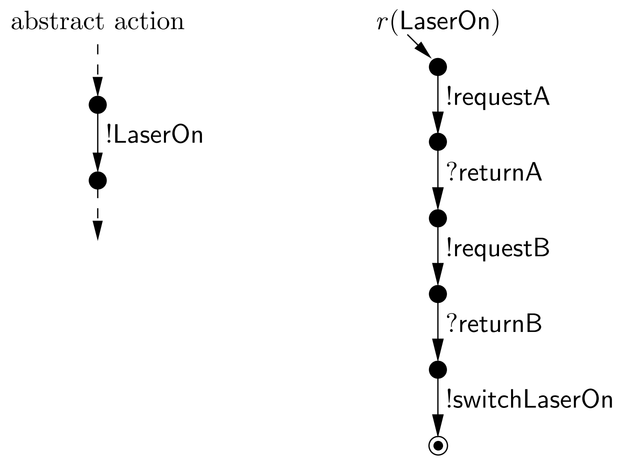


Figure 4.10: Refinement example of parameter abstraction

4.2.5 Abstraction from underlying components

We continue with another remote procedure example from the laser component, this time the initialization of the laser component. The specification prescribes that the call consists of one action, but in the implementation the call consists of several actions that can occur in arbitrary order.

Example 4.2.7 To initialize a component, we send it an init command. When the component is initialized, it sends a ready command back and when it has a problem to initialize, it sends an nready signal. Part of the specification of the component's behavior is shown on the left-hand side of Figure 4.11 (the triangles indicate a continuation of the transition system). On the right-hand side we show a test-case that tests if the IUT returns a ready or nready command after the init stimulus.

Based on the specification it seems to be one component, but it turns out that in the IUT it is implemented as two components A and B. This

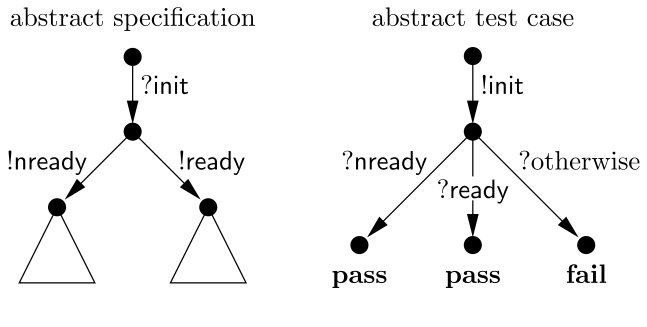


Figure 4.11: Abstract (part of) specification and test-case for the component abstraction example

changes the initialization of the component in the following way: instead of one initialization command, two commands need to be sent (one for each component). After the component has received the initialization command it sends a ready or nready message back. When the components can be initialized in arbitrary order we get an interesting situation. In the most complex case it is the parallel composition of the initialization of both components.

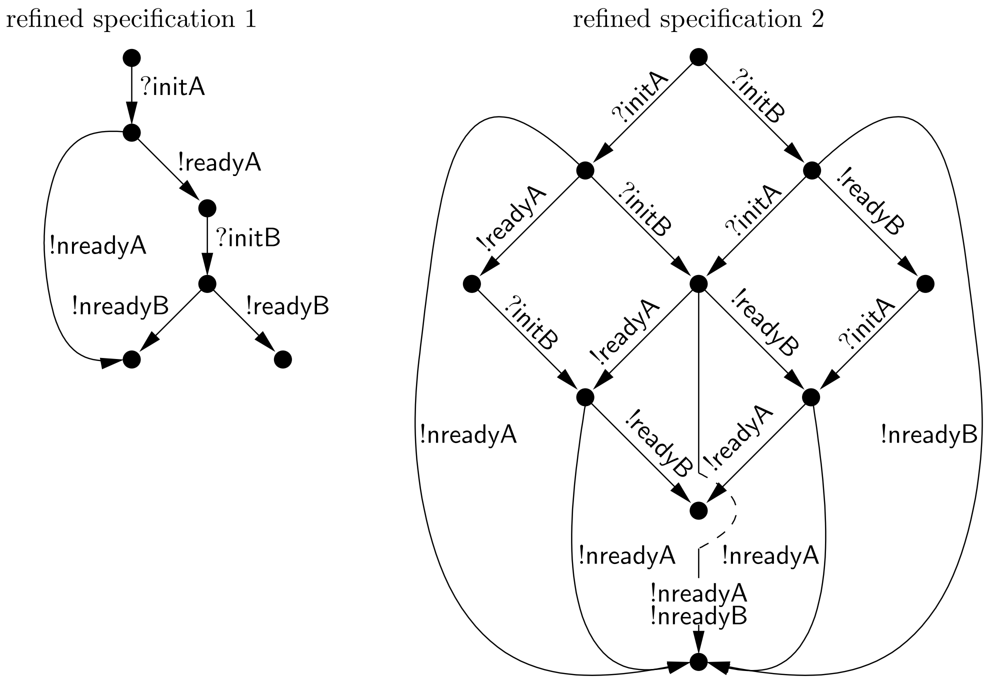


Figure 4.12: Refined specification for component abstraction

In Figure 4.12 we give two possible refined specifications: refined specification 1 initializes component A and B sequentially. We start with initA and when we receive nreadyA we know that the initialization process failed. When we receive readyA we continue with the initialization of component B. We see that there are two possible states after the initialization of the components, one where both components are initialized successfully and one where one or both components are initialized unsuccessfully.

Refined specification 2 is more complicated. In this case we allow components A and B to be initialized in arbitrary order. Like in the abstract case, there are two end states of the initialization process, one for successful and one for unsuccessful initialization (the state at the bottom of the figure is the unsuccessful initialization state and the state at the bottom of the parallelogram indicates successful initialization). Whenever we encounter an nready response of one component, we ignore the nready response of the other component and we go the unsuccessful termination state (another possibility is to also wait for the nready of the other component). When both

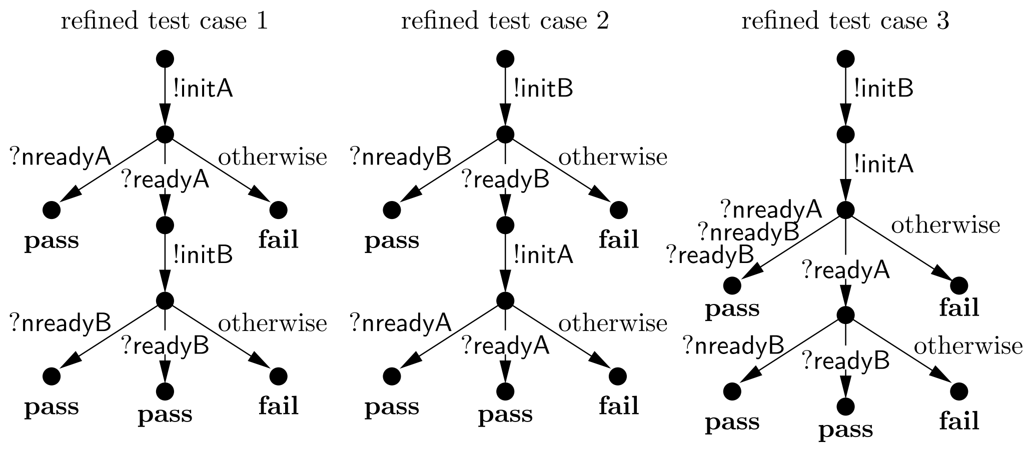


Figure 4.13: Refined test-cases for component abstraction

components are initialized successfully we end in the successful termination state.

For test-cases we see a similar complexity for the refinement, or in some sense it is even more complex. For the case where we allow the components to be initialized in arbitrary order, we need several test-cases to test this behavior. In Figure 4.13 we show three refined test-cases. Test-case 1 corresponds to refined specification 1 from Figure 4.12 on the previous page, test-case 2 and 3 test the behavior of refined specification 2. Test-case 2 tests if the components can be initialized in the order B followed by A. In test-case 3 we first send the initialization commands to A and B and then check the observed responses. □

The required refinements for the last example are not straightforward. We want to express that the abstract action init should be replaced by the independent actions initA and initB, like shown on the left-hand side in Figure 4.14. Likewise we want to express the refinement for ready and nready as shown in the figure. The problem lies in combining the refinements. We cannot simply replace the abstract actions by the behavior in the refinement transition systems. There are dependencies across the refinements, for ex-

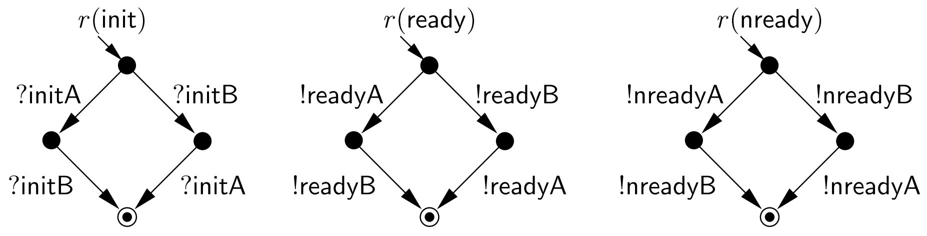


Figure 4.14: Refinement transition systems for component abstraction

ample readyA depends on initA in that it can only occur *after* initA. As we will explain in Section 4.4, dependencies between actions is a weak point of LTS specifications. In order to handle this kind of refinement we need to keep track of the dependencies between refinements.

4.2.6 User interface refinements

A refinement that seems to fit rather nicely with action refinement is what we dubbed "user interface refinement". With this term we want to refer to functionality that is implemented in a system and that needs to be user-accessible via a user interface, for example a web-page. As an example we use the login functionality of an operation system.

On an abstract level, logging into a system may be a direction-less action. In a concrete implementation we need to provide more details, like for example user name and password. The functionality to check a user name - password combination is already there in the system, for example via a remote procedure call. But when we provide this functionality via a user-interface it is used in a different way. For example, the order in which we give the user name and password, may be dictated by the user-interface. The user-interface itself, might also add some behavior, for example if the user repeatedly enters the wrong password, the system might for example block.

Example 4.2.8 In Figure 4.15 on the next page we show on the left-hand side the abstract action login as part of the abstract specification. In the implementation, login is implemented by asking for a user name and password. If these are correct, the login is successful, otherwise we have to reissue our credentials. In the middle of the figure we give the refinement transition system with the required level of detail for login. Based on this information we want to refine an abstract specification and/or test-case. We give an example of a refined test-case on the right-hand side. □

The scenario can be enhanced to make it more realistic. For example, as mentioned above, by taking action if the user logs into the system with the wrong password. After several tries (quite often 3 times) we might want for example to block the user from the system, or prevent the user from logging in for the next hour, etc.

This kind of refinement can be seen in many applications. Take for example an electronic banking system. Operations for bank transactions are already there, only to offer it via a user-interface, the functionality is presented in a user friendly way. For example by asking parameters in a sequential way, or by informing the user via error/information messages.

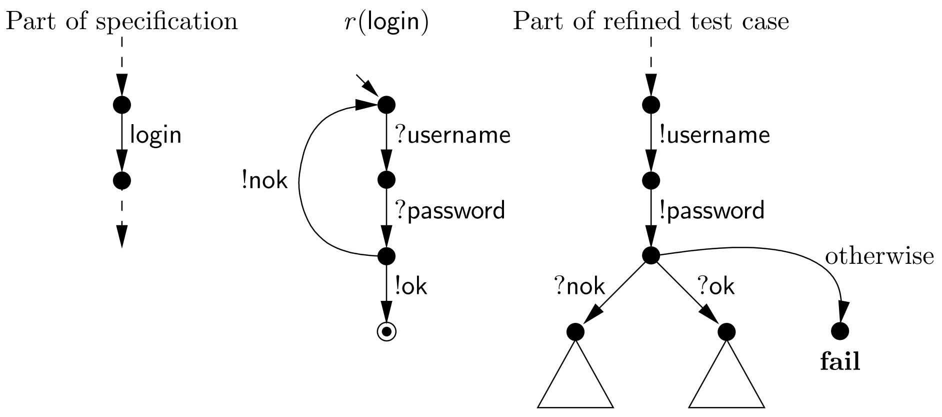


Figure 4.15: Specification, refinement transition system and test-case for login example

4.2.7 Database transactions

Databases are another example where on an abstract level the actions seem simple: create, read, update and delete. When we want to test a database implementation the abstract information is not enough, especially if we want to take transactions with possible aborts into account.

Example 4.2.9 On the left-hand side of Figure 4.16 on the facing page, we show part of a specification that stores some data in a database via the store action. The implementation uses the two-phase commit protocol to ensure atomic transactions. In the middle of the figure we show the refinement transition system for store. First we prepare the transaction with the prepare command. When we receive ok_prepare we continue with commit, and when we receive nok we try again. When commit is followed by ok_commit we are done with the refinement of store and when it is followed by nok we retry the commit action.

□

This example can be extended to incorporate more complex behavior. In most database systems there is a limit on the amount of retries. It is also possible to take transactions into account that consist of other transactions.

4.3 Requirements on action refinement for model-based testing

In Figure 4.17 on page 80 we give an overview of the ingredients of action refinement for model-based testing. On the left-hand side we see the abstract system specification, the refined system specification and the system

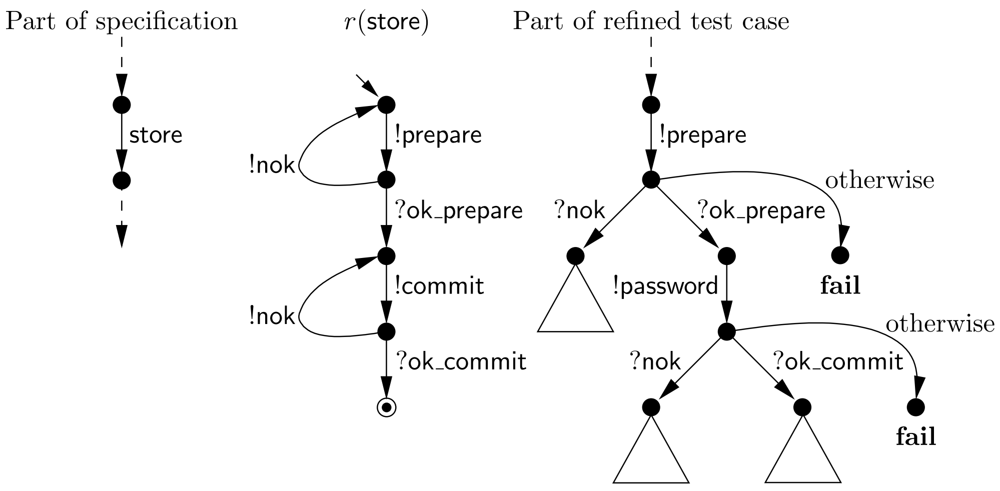


Figure 4.16: Specification, refinement transition system and test-case for database example

implementation. The objects on the right-hand side denote test-suites, also in increasing level of concreteness. From top to bottom, left to right we encounter:

Abstract system specification is a (formal) model of the system implementation. We call it *abstract* to indicate that to use it for testing, we still need to refine it. The specification misses relevant information for testing the implementation.

Refined system specification is the refinement of the abstract system specification, this time with the required level of detail to test the system implementation.

System implementation is the system that we want to test, also known as IUT (Implementation Under Test); it is a real system in the physical world.

Abstract test-suite is the test-suite that is derived from the abstract system specification. As with the abstract system specification, the test-cases lack information to test the system implementation (they are in need of refinement).

Refined test-suite is a test-suite with the required level of detail to test the system implementation. There are potentially two ways to derive such a test-suite. One way is to refine the abstract test-suite, another way is to derive test-cases from the refined system specification.

Executable test-suite is a test-suite in the physical world that we can execute against the system implementation. This results in a verdict

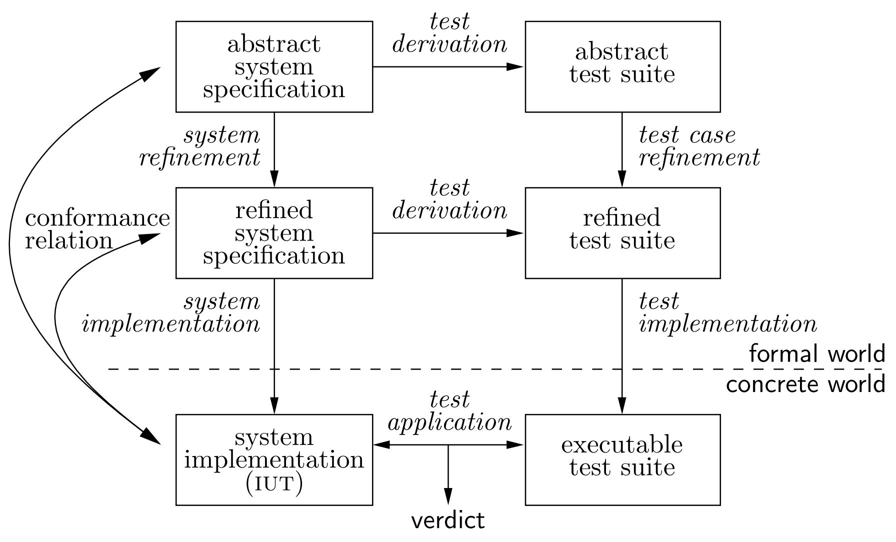


Figure 4.17: Refinement ingredients

whether or not the implementation is correct with respect to the refined (or abstract) system specification.

Central to the research in action refinement for model-based testing is how to obtain refined test-cases; test-cases with the required level of detail. Figure 4.17 shows that there are two ways to obtain a refined test-suite. One way is to refine an abstract specification and derive a test-suite from the refined specification. The other is to refine an abstract test-suite that is generated from an abstract specification. The first approach uses transition system refinement and the other approach uses test-case refinement. We find it an important requirement that these two refinements result in equivalent test-suites. We use completeness with respect to **ioco** as a requirement on our action refinement approach: refinement of a complete test-suite should result again in a refined complete test-suite. Likewise, we should be able to derive a complete test-suite from a refined specification.

Our research questions, in terms of the concepts of the figure, are:

- How, and under what restrictions can we refine system specifications to use them for model-based testing?
- How, and under what restrictions can we refine test-cases?
- Under what circumstances is a refined test-suite, obtained by deriving tests from a refined specification, equivalent to a test-suite obtained by test derivation (from the abstract specification) followed by test-case refinement?

- What conformance relations can we use between the abstract or refined specification and the implementation, and what is the relation between these conformance relations?

4.4 Action refinement results

During our research we have studied the action refinement literature for results that we could use in our own research. In this section we give an overview of action refinement research found in the literature [GR01, vGG89]. We end with an analysis of which parts we can re-use and which parts we have to develop ourselves. We delay the details of *how* to do action refinement to the next chapter, where we treat this in great detail.

We denote the refinement of a single action a to B in A as: $\mathsf{A}[\mathsf{a} \to \mathsf{B}]$, where A and B are behavior expressions, for example expressed as process algebraic terms or transition systems. The next example shows action refinement in operation on transition systems.

Example 4.4.1 Suppose that A is a transition system that consists of two transitions: a followed by b (left-hand side of Figure 4.18). We refine the abstract action a by more complex behavior B. Let B be the transition system in the middle of Figure 4.18: a_1 followed by a_2. The result of $\mathsf{A}[\mathsf{a} \to \mathsf{B}]$ is the system on the right in which the transition with a is replaced with a_1 followed by a_2. □

For quite a while (roughly 1989 - 1995) action refinement enjoyed a broad interest in the process algebra research community. A big part of the research went into looking for equivalence notions to compare models (expressed in some process algebraic language). This is understandable as many contributions to action refinement research took place in *comparative concurrency semantics* research. As the name suggests, comparing models of concurrency is important in this area. The central question in the research was as follows (taken from [GR01]). For action refinement as an operator in

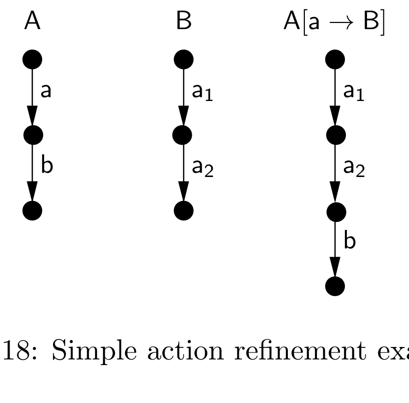


Figure 4.18: Simple action refinement example

a process algebra, given a candidate equivalence notion $\simeq$, we want to find the coarsest relation $\equiv$ contained in $\simeq$ that is a congruence for the operators of the (process algebraic) language. This means mathematically that we are looking for an operator that has the following (congruence) properties:

1. whenever $\mathsf{B} \equiv \mathsf{C}$ then $\mathsf{A}[\mathsf{a} \to \mathsf{B}] \equiv \mathsf{A}[\mathsf{a} \to \mathsf{C}]$;
2. whenever $\mathsf{A} \equiv \mathsf{B}$, then $\mathsf{A}[\mathsf{a} \to \mathsf{C}] \equiv \mathsf{B}[\mathsf{a} \to \mathsf{C}]$.

To put it into words, this means that:

1. When we have two equivalent systems B and C (for example transition system models), then refining an action a in a system A to either B or C should give equivalent refined systems.
2. When we have two equivalent systems A and B, then refining the action a in system A to more refined behavior C should be equivalent with refining the action a to C in system B.

The main requirement for the first part of the congruence question to hold, is that one makes a clear distinction between deadlock (where a system can do nothing at all) and termination (where a system can do nothing but terminate, i.e., relinquish control). The distinction is easily made if one models termination as a special action, or as a special state. We illustrate why we need this distinction with the following example.

Example 4.4.2 Let $\mathsf{A} = \mathsf{a};\mathsf{b}$ and $\mathsf{B}_1 = \mathsf{c}$ (the execution of c leading to successful termination) and $\mathsf{B}_2 = \mathsf{c};\mathsf{0}$ (the execution of c leading to deadlock). B_1 and B_2 are equivalent when ignoring termination, but $\mathsf{A}[\mathsf{a} \to \mathsf{B}_1]$ can perform b after c, while $\mathsf{A}[\mathsf{a} \to \mathsf{B}_2]$ cannot. $\square$

Central to the second part of the congruence issue is *atomicity*. Where the actions in the abstract system are atomic (i.e., indivisible) by default, the question arises whether the same holds when we refine such an indivisible abstract action. We illustrate this issue in the following example.

Example 4.4.3 On the left-hand side of Figure 4.19 on the next page we see an abstract specification with the functionality a followed by b, or b followed by a. In interleaving semantics, $\mathsf{a} \parallel \mathsf{b}$ (a and b happen in *parallel*) is equivalent with $\mathsf{a};\mathsf{b} + \mathsf{b};\mathsf{a}$ (the *choice* between a followed by b or b followed by a). The interesting point is that although they are equivalent, we get different action refinement results for these two situations when we replace action a by a_1 followed by a_2 in the original process terms. We show two possible refinement results in the middle and on the right-hand side of Figure 4.19 on the facing page. In the situation that the actions a and b occur in parallel in the abstract specification, we get the system $(\mathsf{a}_1;\mathsf{a}_2) \parallel \mathsf{b}$.

This is the system on the right-hand side and corresponds to non-atomic refinement. In case of a choice between $\mathsf{a};\mathsf{b}$ and $\mathsf{b};\mathsf{a}$ we get the system in the middle: $(\mathsf{a}_1;\mathsf{a}_2);\mathsf{b}+\mathsf{b};(\mathsf{a}_1;\mathsf{a}_2)$. This corresponds to atomic refinement. In other words, the trace $\mathsf{a}_1{\cdot}\mathsf{b}{\cdot}\mathsf{a}_2$ is a valid trace of the non-atomically refined system, but not of the atomically refined system. □

The example shows two kinds of action refinement: atomic and non-atomic action refinement. In atomic action refinement, we treat the sequence of actions in the refinement again as atomic. In other words, the sequence is indivisible and is executed in an all or nothing manner. This means that no actions can interleave within the refinement. In terms of the example above, when we atomically refine the abstract specification, this means that action b cannot interfere with the actions in the refinement $\mathsf{a}_1;\mathsf{a}_2$ (the refined specification in the middle). With non-atomic refinement, actions are allowed to interfere with the refined actions. In the refined system on the right we see that non-atomic action refinement allows action b to occur between the actions a_1 and a_2.

To be more precise, central to atomic and non-atomic refinement are the *dependencies* between actions. Actions that depend on each other are ordered: when action b depends on action a this means that action b can only occur *after* action a has occurred. When action a and b are independent of each other they can occur in arbitrary order. An example of independent actions are actions that occur in parallel, hence the relation with atomicity. In atomic refinement the sequence of actions in the refinement is considered as one action. In non-atomic refinement the actions in the refinement are considered to be individual actions with individual dependencies. In terms of Example 4.4.3 this means that in the choice scenario '$\mathsf{a};\mathsf{b}+\mathsf{b};\mathsf{a}$', a and b depend on each other. When we refine a into $\mathsf{a}_1;\mathsf{a}_2$, a_1, a_2 and b depend on each other, meaning, b has to wait until a_1 and a_2 are finished and cannot occur after a_1, i.e., *before* a_2. In the parallel scenario '$\mathsf{a}||\mathsf{b}$', a and b are independent of each other. In this case when we refine a into $\mathsf{a}_1;\mathsf{a}_2$, a_1, a_2 and b are independent of each other. This means that b can occur *after* a_1

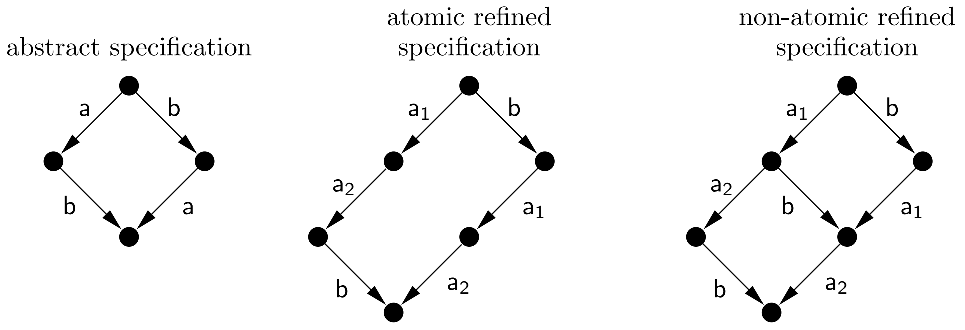

abstract specification
a
b
b
a
atomic refined specification
a1
b
a2
a1
b
a2
non-atomic refined specification
a1
b
a2
b
a1
b
a2

Figure 4.19: Atomic versus non-atomic refinement

and *before* a_2 (this is the situation on the right-hand side of Figure 4.19 on the previous page).

The crux in the discussion about (non) atomic action refinement is *concurrency.* The models of concurrency found in the literature can roughly be distinguished in two kinds: those in which the independent execution of two processes is modeled by specifying the possible interleavings of their (atomic) actions, and those in which the causal relations between the actions of a system are represented explicitly [vGG89]. This is also known as *interleaving* semantics versus *true concurrency* semantics. We refer to [Mon90] for an overview on the many true concurrency semantics. One can define action refinement as an operator on transition systems that is well defined up to strong bisimilarity. However, action refinement as an operator on transition systems does not distribute over parallel composition. This means that action refinement as an operator on transition systems is not usable for non-atomic refinement. It does hold that both strong and weak bisimilarity are congruences for atomic action refinement [DG91].

In order to get a non-atomic refinement operator on labeled transition systems that distributes over parallel composition, one has to impose strong restrictions. When refinement is disallowed for all actions that decide choices, as well as all actions that occur concurrently with themselves strong bisimilarity is a congruence for the resulting operator ([CvGG92]). However, this operator no longer distributes over choice.

It turns out to be rather difficult to use non-atomic action refinement with interleaving semantics. A solution to this problem that has received a lot of attention in the literature is to move to more expressive models than labeled transition systems, in particular so called *true concurrency* semantics. An example of models with true concurrency semantics are *event structures.* As a side step we illustrate how non-atomic action refinement works out for event structures. For this cause we introduce a simplified version of an event structure with enough information to explain the principle.

Definition 4.4.4 [Simplified Event Structure]
A simplified event structure is a 4 tuple $\langle E, \#, \leq, l\rangle$ where

- E is a set of events;
- $\# \subseteq E \times E$ is the conflict relation;
- $\leq \subseteq E \times E$ is the causality relation;
- $l : E \to L$ is the labeling function.

Example 4.4.5 Figure 4.20 on the facing page shows the event structures for $\mathsf{a} \parallel \mathsf{b}$ and $\mathsf{a}; \mathsf{b} + \mathsf{b}; \mathsf{a}$. We can read them in the following way. A node stands for an event, the label next to the node is the label of the event.

An arrow represents the causality relation and a dashed line represents the conflict relation. As we can see, in contrast to the LTS with interleaving semantics from Example 4.4.3, these two event structures are not identical. $\mathsf{a} \parallel \mathsf{b}$ is represented by the events a and b with no causality or conflict relations. This means that the events a and b can be executed independently of each other. $\mathsf{a};\mathsf{b} + \mathsf{b};\mathsf{a}$ is represented by four events. We have $\mathsf{a} \rightarrow \mathsf{b}$ or $\mathsf{b} \rightarrow \mathsf{a}$ with the (top) events a and b in conflict. This reads as: either a followed by b or b followed by a (but not both).

When we refine a into $\mathsf{a}_1;\mathsf{a}_2$ we see the strength of event structures. Figure 4.21 on the next page shows the refined event structures for $(\mathsf{a} \parallel \mathsf{b})[\mathsf{a} \rightarrow \mathsf{a}_1;\mathsf{a}_2]$ and $(\mathsf{a};\mathsf{b} + \mathsf{b};\mathsf{a})[\mathsf{a} \rightarrow \mathsf{a}_1;\mathsf{a}_2]$. In interleaving semantics we would have to make a choice between atomic or non-atomic refinement. In true concurrency semantics the choice is made automatically, because the knowledge of concurrency is now in the model. As a result we get $\mathsf{a}_1;\mathsf{a}_2 \parallel \mathsf{b}$ and $\mathsf{a}_1;\mathsf{a}_2;\mathsf{b} + \mathsf{b};\mathsf{a}_1;\mathsf{a}_2$, as conforms to our intuition. □

It turns out that isomorphism of event-based models gives rise to a congruence. However, this relation is rather strong and a large part of the literature on action refinement is devoted to the quest for alternative congruences, in particular ones that are weaker (less distinguishing), like *history-preserving bisimulation* (see for example [vGG89]). Another approach is to use less distinguishing models than event-based models, for example *causal trees* ([DD89, DD93]). To obtain the coarsest congruence, the minimal amount of information one must add is to distinguish related beginnings and endings of all actions. This is called the *ST*-principle, introduced by Van Glabbeek and Vaandrager [vGV87]. *ST-bisimilarity* is an equivalence relation based on the ST-principle and the coarsest congruence contained in strong bisimilarity.

Related to atomicity is the distinction between syntactic and semantic refinement, which for quite a while was seen as an important dichotomy in action refinement research. Syntactic refinement refers to refinement on the syntax level of a process algebra, whereas semantic refinement refers to refinement on the level of the semantics of a process algebra (for example LTS SOS rules). For atomic refinement both approaches give similar results. For non-atomic refinement, semantic refinement turns out to be more appropriate, because dealing with concurrency is more difficult for syntactic action refinement.

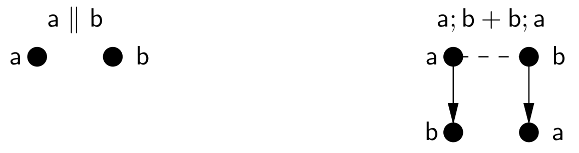


Figure 4.20: Example of an event structure

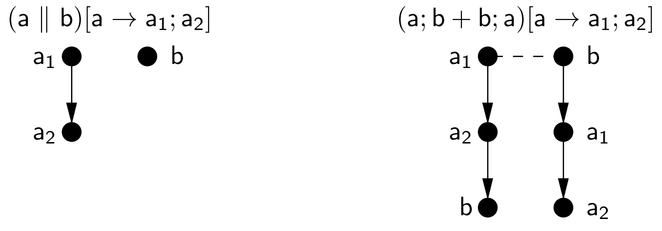


Figure 4.21: Refinement on event structures

This is the quest for the coarsest congruence in a nutshell. We will not treat it in more detail, because for our research and for this thesis, we do not reuse much of the results of this research. We encourage the interested reader to take a look at the cited references. A lot of effort in the action refinement research went into the quest for the coarsest congruence. After this puzzle was solved the field has been deserted again. This is a pity because the research after the coarsest congruence is not directly applicable for our research. The research found in the literature, like [GR01], provides a good basis for the atomic refinement of transition systems. How to refine test-cases, how to treat inputs, outputs and the absence of outputs are examples of interesting problems that we need to investigate.

4.4.1 Relevance for model-based testing

What part of the research on action refinement can we re-use for our test oriented research? Which parts do we have to solve ourselves? What are the specific characteristics of our domain? To start with the last one, we identify inputs, outputs and quiescence.

Inputs, outputs and quiescence. The theories found in the action refinement literature do not take inputs, outputs and quiescence into account (the actions do not have direction). This means that we have to take care of this ourselves. In general we can reuse the approaches found in the literature for transition system refinement, but we have to be careful not to introduce or break quiescence in our refinements.

Example 4.4.6 In Figure 4.22 on the facing page we show an abstract system that consists of one input action: a (left-hand side). An abstract test-case to test this system first does an observation to see that the system is quiescent and then performs a. When we refine a to $\mathsf{a}_1; \mathsf{a}_2$, where a_1 and a_2 are output actions, δ is no longer a valid first observation. This may lead to unwanted results. We explain in Chapter 5 how to handle this situation. □

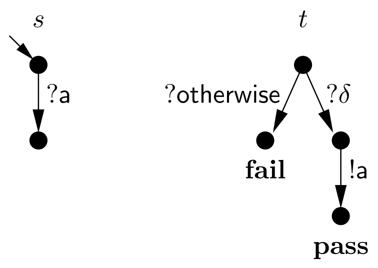


Figure 4.22: Quiescence in test-cases

Labeled transition systems. As we have shown in Section 2.5, the **ioco** test theory is based on labeled transition systems. This means that non-atomic action refinement is difficult. There is some theory for non-atomic action refinement on LTSs, but these theories are quite complex and put heavy restrictions on the types of refinements.

An option would be to adapt the **ioco** theory to event structures. This is an interesting idea, however there are some limitations to event structures. For example, it is hard, if not impossible to express infinite behavior in a finite way, like the way loops work in an LTS. Another issue is that our test-tooling is based on LTSs. This would mean that we have to change our test-theory and our tooling.

Test cases. Action refinement is not defined for test-cases. We cannot reuse the existing theory, because of the special nature of test-cases. For example, in an observation step, a test-case observes all outputs of the IUT. In the refined test-case, we have to observe all refined output actions. We do not get these with the standard action refinement theory. Another interesting point is that, where the refinement of a transition system leads to one refined transition system, the refinement of a test-case may lead to several refined test-cases, as we show in the next example.

Example 4.4.7 Let us look again at the abstract test-case of our video game example in Figure 4.1 on page 66: we input three euro, we press the play button and we expect to play a game. Suppose that we refine i€3 to i€1 followed by i€2, or i€2 followed by i€1. This means that instead of the action i€3 we can think of several test steps to test the refined behavior: we can enter one euro followed by two euro, or vice versa, we can perform a refund in between, etc. In other words, the refinement of the abstract test-case may result in more than one test-case! □

4.5 Action refinement classification

In order to talk about action refinement in model-based testing, it is helpful to have an idea of the relevant concepts that play a role. We have already seen some concepts in Section 4.4, for others we were inspired by the scenario's in Section 4.2. In this section we propose an action refinement classification. It is a pragmatic classification and we do not claim it to be complete. The concepts were developed during our research based on the scenarios and problems that we encountered. Our action refinement classification focuses primarily on the behavior of the refinement. As far as we know this is new. Our classification consists of the properties: *atomicity*, *linearity*, *boundedness*, *initiative* and *observability*. We have used this classification to direct our research and we use it to position our action refinement research of Chapter 5 and to make the possibilities and limitations clear.

Atomicity is an important concept in action refinement. In atomic refinement we treat the refined behavior again as atomic, i.e., indivisible, whereas in non-atomic refinement this is not the case. We discussed the concept atomicity in Section 4.4. As we have seen in the previous section, the intuition behind refinement of LTSs is that transitions in the abstract system or test case are replaced by more complex behavior. We illustrated in Example 4.2.6 that this approach does not work for non-atomic refinement. We treat atomic refinement in more detail in Chapter 5.

Linearity expresses whether branching occurs in the refinement behavior. In case of linear refinement this is not the case, and in case of non-linear or branching refinement it is. The navigation scenario of Example 4.2.3 is an example of linear refinement, whereas the refinement of output action $_o$€3 in Figure 4.1 on page 66 is an example of refinement with branching behavior. We find linearity an important concept, because composition plays an important role in action refinement research (especially for LTS-based systems). Non-linear behavior indicates behavior that is the result of composition (parallelism or choice).

Boundedness refers to bounded or finite behavior. In case the refinement behavior is bounded, this means that the refinement behavior ends in a finite number of steps. Loops in a refinement introduce unbounded, or potentially infinite behavior. The refinement of input action $_i$€3 in Figure 4.3 on page 68 (treated in Example 4.2.2) is an example of unbounded behavior. When we always press the refund button after inserting the one or two euro coin, we create infinite behavior without "leaving" the refinement transition system. For LTSs, unboundedness may seem trivial, because they support it in a natural way. To action refinement in general it is important, because for example event structures do not support unbounded behavior (not in a finite

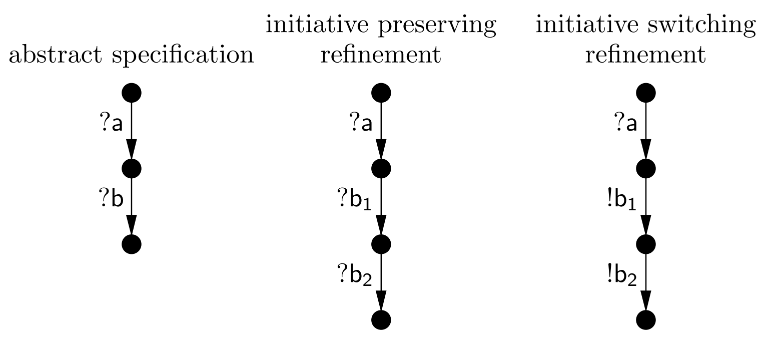

abstract specification
initiative preserving refinement
initiative switching refinement
?a
?b
?a
?b1
?b2
?a
!b1
!b2

Figure 4.23: Preservation of initiative example

way).

Initiative refers to the direction of the action, i.e., input, output or no direction. Initiative preserving refinement of input actions means that all transitions starting in the start state of the refinement transition system are input actions (this also means that it does not start with an internal action). For output refinements, preservation of initiative means that at least one of the transitions starting in the start state is an output action. For the input actions we need all transitions to start with an input action, because it takes only one output action to destroy quiescence. For the output actions it is the other way around. We need at least one transition to start with an output, because this preserves non-quiescence. If none of the transitions start with an output action, this might introduce quiescence where previously there was none. Preservation of initiative has the result that quiescence of the abstract system is preserved in the refined system. We illustrate this in the following example, see also Example 4.4.6.

Example 4.5.1 In Figure 4.23 we see an abstract specification (left-hand side) where we can do a followed by b; both are input actions. We refine a to itself and b to $b_1; b_2$. We consider two cases: one where b_1 and b_2 are input actions (middle), the other where b_1 and b_2 are output actions (right-hand side). The system in the middle preserves the initiative: in the abstract system we have the input action b and in the refined system the input action b_1. The system on the right-hand side does not preserve the initiative, because b_1 is an output action (the initiative is *switched* from input to output). Preservation of initiative is important for *quiescence*. We see that the abstract system s is quiescent after a: $\delta \in out(s \textbf{ after } \mathsf{a})$. With initiative preserving behavior we keep this behavior. $\delta \in out(s[r] \textbf{ after } \mathsf{a})$, where a is a trace of $r(\mathsf{a})$. With refinements that do not preserve the initiative, like the one on the right-hand side, we lose this behavior: $\delta \notin out(s[r] \textbf{ after } \mathsf{a})$ $(= \{\mathsf{b}_1\})$. □

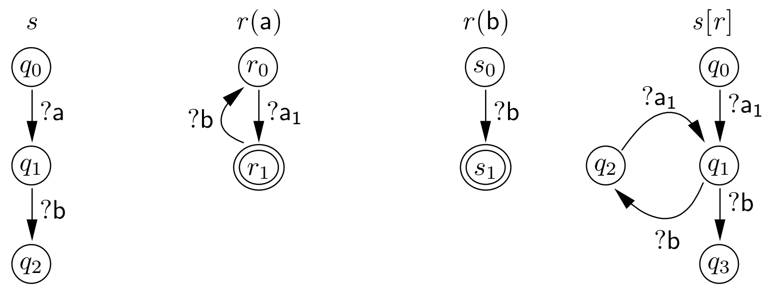


Figure 4.24: Observability example

Quiescence is an important concept in the **ioco** theory. Therefore we find preservation of initiative an important property. In case the initiative is not preserved, it is immediately clear that we cannot use **ioco**.

Observability in action refinement means whether we can distinguish refined behavior from abstract behavior in the refined system. In particular do we know when the refined behavior (successfully) terminates? When we do not know if the refinement ended or not, we can run into difficulties relating refined behavior to abstract behavior, and vice versa, as we will illustrate with the following example.

Example 4.5.2 In Figure 4.24 we show the refinement of a system that can perform a followed by b. We refine a into a system that can perform a_1 followed by b followed by a_1 ad infinitum. We refine b to itself, i.e., the action is unchanged. The resulting refined system is depicted on the right. The issue with this system from an observability point of view is that in the refined system we may not know when refined behavior stops, based only on knowledge of the refined system. Does the execution $q_0\mathsf{a}_1q_1\mathsf{b}q_2$ originate from the refinement of a or a·b? If it is possible to tell where the execution originated from, we say that the refinement is observable, otherwise it is unobservable. □

4.6 Atomic action refinement in model-based testing

When we set out with our research, our goal was to have as little restrictions as possible on the types of refinements that we allow. In this section we explain the constraints that we choose and the reasons behind them.

In terms of the classification from Section 4.5, our action refinement approach is *atomic*, allows *non-linear* and *unbounded* behavior, and restricts to observable refinements.

The restriction on atomicity is a pragmatic one. As we discussed in

Section 4.4, non-atomic refinement on transition systems is quite difficult and cumbersome. We want to reuse the **ioco** theory and therefore we decided to focus on atomic action refinement for the **ioco** test theory. Hence we also need the restriction on initiative in order to relate quiescence in the abstract and refined case.

We can express our constraints in a more formal manner on the type of refinement function that we allow. We start with the formal definition of the constraint followed by an explanation in words. Let $r : L_\tau \to \mathbf{FLTS}$ be a refinement function.

Definition 4.6.1

1. Preservation of input initiative. Let $a \in I$, and let $r(a)$ be the refinement transition system of a, then

   $$init(r(a)) \subseteq I_{r(a)} \tag{4.1}$$

   This means that the refinement transition system of an input action is only allowed to start with an input action (no output action or τ action).

2. Preservation of output initiative. Let $x \in U$, and let $r(x)$ be the refinement transition system of x, then it should hold that

   $$\mathsf{start}_{r(x)} \overset{\tau}{\not\to} \wedge \exists y \in U_{r(x)} : \mathsf{start}_{r(x)} \xrightarrow{y} \tag{4.2}$$

   This means that the refinement transition system of an output action should start with at least one output action and is not allowed to start with an internal action.

3. Internal actions remain internal actions.

   $$r(\tau) = \langle\{\mathsf{start}, \mathsf{final}\}, \emptyset, \emptyset, \{(\mathsf{start}, \tau, \mathsf{final})\}, \mathsf{start}, \mathsf{final}\rangle \tag{4.3}$$

   The refinement transition system of τ is fixed. This is because we do not want abstract internal actions to become observable in the refined system.

4. No forgetful refinement.

   $$\forall \mu \in L : \mathsf{start}_{r(\mu)} \overset{\epsilon}{\not\Rightarrow} \mathsf{final}_{r(\mu)} \tag{4.4}$$

   No refinement transition system, except $r(\tau)$, can perform the empty trace between the start and final state. This property is called forgetful refinement [vGG01] (forgetting actions by replacing them with the empty trace).

5. No outgoing transitions in the final state. Let $\mu \in L_\tau$, $r(\mu)$ be the refinement transition system of μ, then

$$init(\mathsf{final}_{r(\mu)}) = \emptyset \tag{4.5}$$

This means that the final state of a refinement transition system has no outgoing transitions. With this constraint we know that final signals the end of a refinement.

An alternative for constraint 3 is to let the refinement function only range over L and to preserve internal actions in the way systems and test-cases are refined. An alternative for constraint 5 is to add the constraint of no outgoing transitions in the final state to the FLTS definition. We come back to these restrictions in Section 5.6 and illustrate the consequences of the restrictions in terms of the results of Chapter 5.

An implicit constraint is that the FLTS has only one final state. This means that the refined behavior ends at one place and it is not possible to model refinements that have several successful endings. This is an interesting area for further research. To lift this constraint, one has to find a way to relate the multiple end-states of the refinement transition system to the abstract system.

With these restrictions we are able to tackle most of our scenarios. In Table 4.1 we show the classification of the scenarios that we treated in Section 4.2. On the left-hand side we show the scenarios and on the top we show the classes. In order to fit the table on the page we used the following abbreviations: *A* for atomic and *NA* for non-atomic, *L* for linear and *NL* for non-linear, *B* for bounded and *UB* for unbounded, *IP* for initiative-preserving and *ID* for initiative destroying, *O* for observable and *NO* for not-observable. This works as follows: the linear output splitting example of Section 4.2.2 is classified as atomic, linear, bounded and observable.

For the RPC and user interface scenario we put a question mark for initiative. It depends on the initiative of the abstract action whether the refinement is initiative preserving or not, but in the scenario we abstracted from the initiative. The scenario is initiative preserving if LaserStatus is seen as an output action.

The component abstraction example is the most complex scenario. The case where the initialization of components A and B can occur in arbitrary order is an example of non-atomic refinement, it is non-linear (there is choice involved), bounded (no loops) and initiative preserving. It is not clear if the scenario is an example of observable refinement. For example with which part of the abstract system does the trace initA·readyA correspond? Did we already complete the abstract actions init and ready?

The scenario that we cannot handle with atomic action refinement is the initialization scenario of Section 4.2.5, which requires non-atomic refinement. Refinements of abstract actions without direction, can be handled as follows.

criterion / scenario	atomic	linear	bounded	initiative	observable
Linear output splitting, Section 4.2.2	A	L	B	IP	O
Calculator, Section 4.2.3	A	NL	UB	IP	O
RPC, Section 4.2.4	A	L	B	?	O
Component abstraction, Section 4.2.5	NA	NL	B	IP	?
User interface, Section 4.2.6	A	NL	UB	?	O
Database, Section 4.2.7	A	NL	UB	IP	O

Table 4.1: Classification of action refinement scenarios

If an output action is possible in the start state of the refinement transition system, we interpret the abstract action as an output action, otherwise we interpret it as an input action. In this manner the refinement is initiative preserving.

4.7 Conclusion

In this chapter we introduced action refinement in conformance testing. We started with a gentle introduction into action refinement in model-based testing by discussing several scenarios that we collected during our research. We continued with a discussion of the action refinement ingredients that play a role in our research. Using this as our inspiration we investigated what results from the action refinement research found in the literature we could re-use. We concluded that the existing research on action refinement is a good starting point, but that action refinement for model-based testing requires more. Especially the fact that the **ioco** theory identifies inputs and outputs and that test-cases differ fundamentally from regular transition systems are issues that need to be taken care of. We combined the phenomena described in the scenarios with the action refinement results in an action refinement classification. We used this classification to describe the constraints that we put on our action refinement approach that we discuss in the next chapter: atomic action refinement.

Looking back, we are a bit surprised about the nature of the existing action refinement research. It seems like most of the work done in this area is related to what we called the quest for the coarsest congruence. When one looks in other directions there is less work done. It seems to us that action refinement is a technique that can be of good use in practice. We can imagine that action-refinement-like behavior can help programmers.

Take for example refactoring support in an IDE (Integrated Development Environment) as example. However there is little or no research in this direction.

Considering the amount of research done in this area, it was surprisingly hard to find action refinement scenarios. It seems like we are (one of) the first to look at action refinement from a practical perspective. Therefore we expect our action refinement scenarios to be a contribution to the field.

When we ran into the problem of refinement with a label-set that distinguished inputs and outputs we wondered if there would be more issues that had to taken care of. It would have been very helpful if there were some kind of action refinement classification that showed the (im)possibilities of the action refinement operation itself. Because we could not find such a classification, we defined one ourselves.

All in all, we find it a pity that the action refinement research field has been deserted again after the quest for the coarsest congruence was finished. We think the computer science community would profit from research and proper tooling to support top-down design.

Chapter 5

Using atomic refinement to obtain refined test-cases

In this chapter we present a theory to atomically refine test-cases and transition systems. We show that deriving a refined test suite by first refining the model and then generating the test suite, or by refining the test suite immediately, are equivalent. This work is based on [vdBRT07]. Our earlier work on action refinement (see [vdBRT05]) is superseded by this work, since the case that we studied in [vdBRT05], input-input refinement, is a special case of the atomic action refinement theory that we discuss in this chapter.

5.1 Introduction

The previous chapter was the gentle introduction to action refinement in model-based testing. In this chapter we provide the definitions and theorems that we use for atomic action refinement. The chapter is structured in the following way. We start with transition system refinement in Section 5.2, followed by trace refinement in Section 5.3. In Section 5.4 we present $\mathbf{ioco}_r$, an implementation relation between the abstract specification and the implementation that takes refinement into account, followed by test-case refinement in Section 5.5. We revisit our constraints in Section 5.6 and we conclude with Section 5.7 and directions for further research in Section 5.8. Although we do not always state so explicitly, we assume throughout this chapter, that all used refinement functions satisfy the constraints of Section 4.6.

5.2 Transition system refinement

We refine transition systems using Definition 5.2.1. $(\mathsf{start}, \checkmark)$ is a special state to mark the start state of the refined system. $\checkmark$ is a special state and is assumed not to occur in any of the other state sets. Final denotes the set

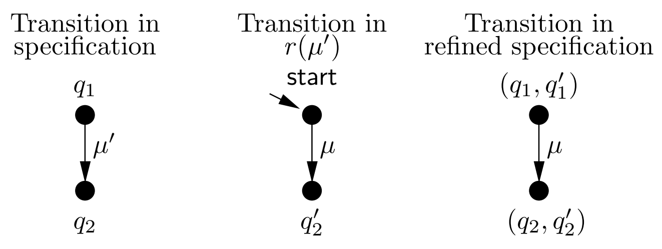


Figure 5.1: Example of transitions in T_1

of all final states of all refinement transition systems: $\mathsf{Final} = \{\mathsf{final}_{r(\lambda)} \mid \lambda \in L_\tau\} \cup \{\checkmark\}$.

Definition 5.2.1 Let $s = \langle Q, I, U, T, \mathsf{start}\rangle$, $r : L_\tau \to \mathbf{FLTS}$. We assume that the states of the refinement transition systems are all new: $\forall \mu_1, \mu_2 \in L_\tau : \mu_1 \neq \mu_2 \Rightarrow (Q \cap (Q_{r(\mu_1)} \cup Q_{r(\mu_2)})) = \emptyset \wedge Q_{r(\mu_1)} \cap Q_{r(\mu_2)} = \emptyset$. The refined system $s[r]$ is defined as follows. $s[r] = \langle Q_r, I_r, U_r, T_r, \mathsf{start}_r\rangle$:

$$\begin{array}{lcl}
Q_r & = & (Q \times \bigcup_{\mu\in L_\tau} Q_{r(\mu)}) \cup \{(\mathsf{start}, \checkmark)\} \\
I_r & = & \bigcup_{\mu \in L} I_{r(\mu)} \\
U_r & = & \bigcup_{\mu \in L} U_{r(\mu)} \\
T_1 & = & \{((q_1, q_1'), \mu, (q_2, q_2')) \mid q_1' \in \mathsf{Final} \wedge \exists \mu' \in L_\tau : (q_1, \mu', q_2) \in T \\
 & & \qquad\qquad\qquad\qquad \wedge\, (\mathsf{start}_{r(\mu')}, \mu, q_2') \in T_{r(\mu')}\} \\
T_2 & = & \{((q_1, q_1'), \mu, (q_1, q_2')) \mid \exists q \in Q, \mu' \in L_\tau : (q, \mu', q_1) \in T \\
 & & \qquad\qquad\qquad\qquad \wedge\, (q_1', \mu, q_2') \in T_{r(\mu')}\} \\
T_r & = & T_1 \cup T_2 \\
\mathsf{start}_r & = & (\mathsf{start}, \checkmark)
\end{array}$$

The definition reads as follows. The set of states Q_r is the Cartesian product (state tuples) of the set of abstract states with the set of states from the refinement transition systems, plus the start state of the refined system: $(\mathsf{start}, \checkmark)$. The input label set is the union of all input label sets of the refinement transition systems and the output label set is the union of all output label sets. The crux of the definition is in T_1 and T_2. T_1 takes care of all the first transitions in a refinement transition system and T_2 takes care

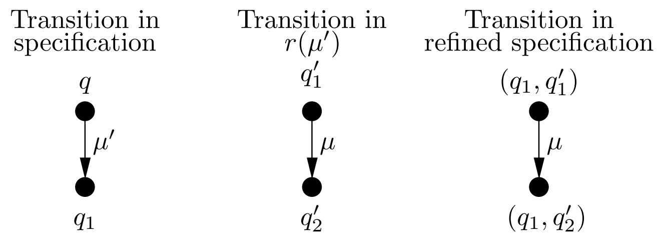


Figure 5.2: Example of transitions in T_2

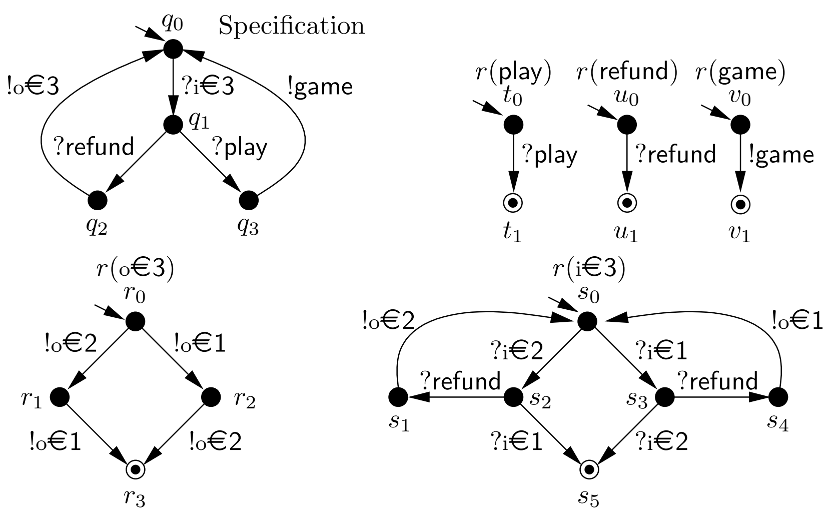


Figure 5.3: LTS refinement: specification and refinement transition systems

of all the other transitions. In Figure 5.1 on the facing page and Figure 5.2 on the preceding page we show how transitions in T_1 and T_2, respectively, are created. We used the same variable names as in the definition for easy reference. Figure 5.1 on the facing page shows transition (q_1, μ', q_2) in the abstract transition system on the left-hand side. In the middle we show transition $(\mathsf{start}, \mu, q_2')$ of refinement transition system $r(\mu')$. For final state q_1' we add transition $((q_1, q_1'), \mu, (q_2, q_2'))$ on the right-hand side to T_1. In a similar fashion we illustrate transition set T_2 in Figure 5.2 on the preceding page. In the following example we show how an entire transition system is refined.

Example 5.2.2 In Figure 5.3 we show the abstract specification of the video game (top, left-hand side) with the refinement transition systems of the abstract actions. To keep the figures readable we will refine the system

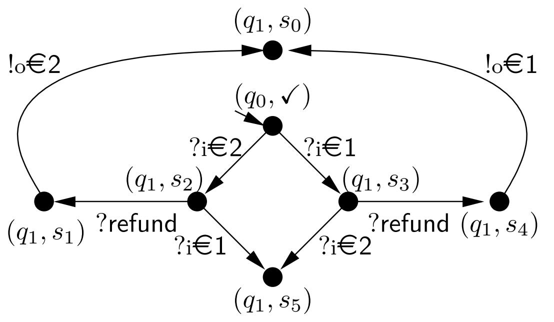


Figure 5.4: LTS refinement step 1

in two steps. First we refine the abstract action i€3, after that we add the rest. We show part of the result of LTS refinement in two steps, the first step in Figure 5.4 on the previous page and the second step in Figure 5.5 on the facing page.

We start with refining the transition from the start state in the abstract system: $q_0 \xrightarrow{\text{i€3}} q_1$. This results in the first transitions in the refined system from its start state $(q_0, \checkmark)$. As $\checkmark \in \mathsf{Final}$, we can add two transitions from $r(\text{i€3})$, according to T_1: $(q_0, \checkmark) \xrightarrow{\text{i€2}} (q_1, s_2)$ and $(q_0, \checkmark) \xrightarrow{\text{i€1}} (q_1, s_3)$. We continue with transitions from (q_1, s_2). As $s_2 \notin \mathsf{Final}$, T_1 does not apply. According to T_2 we can add two transitions: $(q_1, s_2) \xrightarrow{\text{refund}} (q_1, s_1)$ (because $(s_2, \text{refund}, s_1) \in T_{r(\text{i€3})}$) and $(q_1, s_2) \xrightarrow{\text{i€1}} (q_1, s_5)$. When we add all the transitions for $r(\text{i€3})$ we obtain a transition system as is shown in Figure 5.4 on the previous page. Note that transition $(q_1, s_1) \xrightarrow{\text{o€2}} (q_1, s_0)$ does not go back to the start state $(q_0, \checkmark)$, although it goes back to the start state s_0 in the refinement transition system $r(\text{i€3})$. This technique is known as root unwinding. Suppose that in q_0 we could also do action play besides i€3 and suppose that we would cycle back to $(q_0, \checkmark)$. Then the refined actions of play are enabled again, whereas we had already chosen for the refined actions of i€3. When we also add all the other transitions we get the system as depicted in Figure 5.5 on the facing page. □

Note that the refined video game system in this form looks quite different than the one we showed in the previous chapter in Figure 4.2 on page 67. This is primarily the result of the way we refine transition systems. It seems like the resulting transition system can be made smaller. In our video game example above, the states (q_1, s_0), $(q_0, \checkmark)$, (q_0, v_1) and (q_0, r_3) could be taken together.

Note that LTS refinement also creates some unreachable states and transitions that we did not depict in our figures. This is because we take the Cartesian product of the state sets. This creates too many states; an example of such a state is (q_0, t_1) in the previous example. T_1 and T_2 may add transitions for these states. They do not form a problem because they are not reachable and therefore they can be deleted.

5.3 Trace refinement

In order to relate abstract traces to traces of a refined system, we need some form of trace refinement. The operationalization of trace refinement uses two special sets of suspension traces of refinement transition systems, so called *XStraces* (Special Straces) and *TXStraces* (Terminating Special Straces).

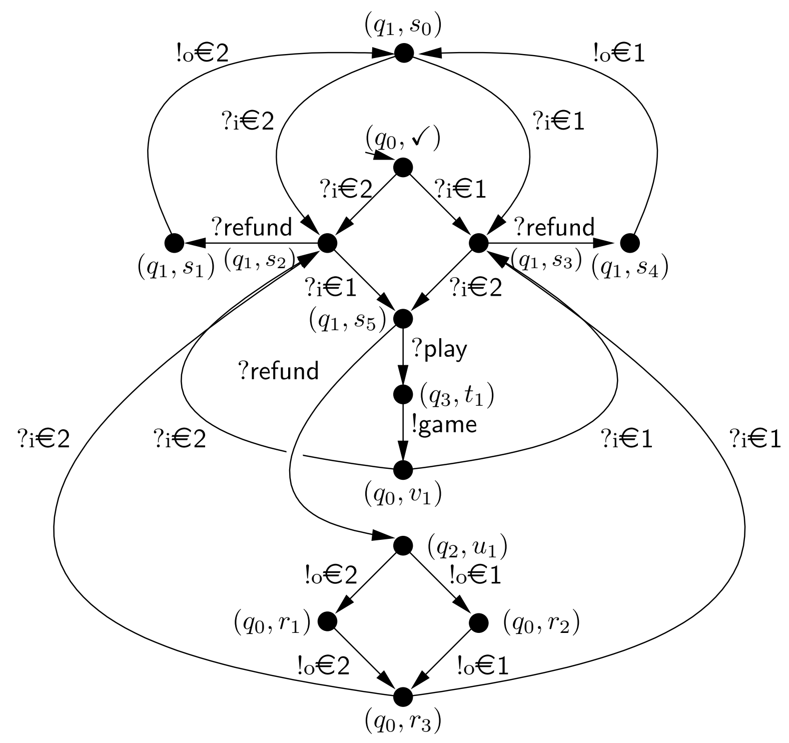


Figure 5.5: LTS refinement step 2

Definition 5.3.1 [Terminating Special Straces] Let $s \in \mathbf{FLTS}(I, U)$

$$TXStraces(s) = \{\sigma \in L_\delta^* \backslash (\delta{\cdot}L_\delta^* \cup L_\delta^*{\cdot}\delta) \mid \mathsf{start} \overset{\sigma}{\Longrightarrow} \mathsf{final}\}$$

Definition 5.3.2 [Special Straces] Let $s \in \mathbf{FLTS}(I, U)$

$$XStraces(s) = \{\sigma \in L_\delta^* \backslash ((\delta{\cdot}L_\delta^*) \cup \{\epsilon\}) \mid \exists q \in Q \backslash \{\mathsf{final}\} : \mathsf{start} \overset{\sigma}{\Longrightarrow} q\}$$

Special Straces are non-empty suspension traces of the refinement transition system that do not *start* with δ and do not *end* in the final state. Terminating Special Straces are suspension traces of the refinement transition system, that do not start, nor end with δ and that end in the final state.

We identify two kinds of trace refinements: complete and incomplete refinements. Complete refinements end in a final state of a refinement transition system versus incomplete refinements ending in a non-final state (it can be the case that a refined trace is in both sets).

The refinement function is not defined for δ and hence there is no refinement transition system for δ. In order to keep our definitions compact, we want to treat δ as all other actions. Therefore we explicitly define $TXStraces(r(\delta)) = \{\delta\}$ and $XStraces(r(\delta)) = \emptyset$.

Definition 5.3.3 [Complete atomic trace refinement]
Let $\sigma = \lambda_1 \cdots \lambda_n, n \geq 0, \forall 1 \leq i \leq n : \lambda_i \in L_\delta, r : L_\tau \to \mathbf{FLTS}$.

$$\sigma[r]_{rc} = \begin{cases} \{\epsilon\} & \text{if } n = 0 \\ \{\sigma_1 \cdots \sigma_n \mid \forall 1 \leq i \leq n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))\} & \text{if } n > 0 \end{cases}$$

We see that complete refinement of a trace means concatenation of all possible *TXStraces* of the individual actions of the trace.

Definition 5.3.4 [Incomplete atomic trace refinement]
Let $\sigma = \lambda_1 \cdots \lambda_n$, $n \geq 0$, $\forall 1 \leq i \leq n : \lambda_i \in L_\delta$, $r : L_\tau \to \mathbf{FLTS}$.

$$\sigma[r]_{inc} = \begin{cases} \emptyset & \text{if } n = 0 \\ \{\sigma_1 \cdots \sigma_n \mid \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)), & \\ \qquad\qquad\qquad \sigma_n \in \mathit{XStraces}(r(\lambda_n))\} & \text{if } n > 0 \end{cases}$$

Incomplete refinement of a trace is similar to complete refinement, except that the final trace is in *XStraces* instead of *TXStraces*. The general definition of atomic trace refinement takes the union of complete and incomplete refinements.

Definition 5.3.5 [Atomic trace refinement] Let $\sigma \in L_\delta^*, r : L_\tau \to \mathbf{FLTS}$.

$$\sigma[r] = \sigma[r]_{rc} \cup \sigma[r]_{inc}$$

We extend the refinement definitions for sets of traces: $\Sigma[r]_{rc} = \bigcup_{\sigma \in \Sigma} \sigma[r]_{rc}$, $\Sigma[r]_{inc} = \bigcup_{\sigma \in \Sigma} \sigma[r]_{inc}$ and $\Sigma[r] = \bigcup_{\sigma \in \Sigma} \sigma[r]$ for $\Sigma \subseteq L_\delta^*$,

Example 5.3.6 To illustrate trace refinement we use our video game system with its refinements in Figure 5.3 on page 97. Suppose that we want to refine the (abstract) trace $_\mathrm{i}$€3·refund·$_\mathrm{o}$€3. We start with the complete refinement of the trace. Definition 5.3.3 shows that complete trace refinement is a concatenation of *TXStraces*. For $_\mathrm{i}$€3 we take the set *TXStraces*(r($_\mathrm{i}$€3)). To keep this example concise we only show two elements of the set of *TXStraces*(r($_\mathrm{i}$€3)): $_\mathrm{i}$€2·$_\mathrm{i}$€1 and $_\mathrm{i}$€1·refund·$_\mathrm{o}$€1·$_\mathrm{i}$€2·$_\mathrm{i}$€1. The refinement transition system for refund is straightforward with *TXStraces*(r(refund)) = {refund}. *TXStraces*(r($_\mathrm{o}$€3)) = {$_\mathrm{o}$€1·$_\mathrm{o}$€2, $_\mathrm{o}$€2·$_\mathrm{o}$€1}. Combined, we can see that $_\mathrm{i}$€2·$_\mathrm{i}$€1·refund·$_\mathrm{o}$€1·$_\mathrm{o}$€2 and $_\mathrm{i}$€1·refund·$_\mathrm{o}$€1·$_\mathrm{i}$€2·$_\mathrm{i}$€1·refund·$_\mathrm{o}$€2·$_\mathrm{o}$€1 are examples of completely refined traces. Incompletely refined traces are constructed analogously, where for $_\mathrm{o}$€3 we take the set *XStraces*(r($_\mathrm{o}$€3)) = {$_\mathrm{o}$€1, $_\mathrm{o}$€2}. □

Example 5.3.6 shows why we forbid *TXStraces* to start and end with δ: a refinement transition system does not provide enough information to decide if quiescence is appropriate at this place. For example if we allow *TXStraces*(r(refund)) to end with δ we get an incorrect refinement, as this results in the erroneous trace $_\mathrm{i}$€2·$_\mathrm{i}$€1·refund·δ·$_\mathrm{o}$€1·$_\mathrm{o}$€2 $\in$ ($_\mathrm{i}$€3·refund·$_\mathrm{o}$€3)[r] (δ cannot be followed by an output action such as $_\mathrm{o}$€1 in this case).

Trace contraction. We call the inverse of refinement *contraction.* Similar to trace refinement we define complete, incomplete and general trace contraction. We use the notation L_r to refer to the union of label sets of the refinement transition systems (for a certain refinement function). Formally, for a refinement function $r : L_\tau \to \mathbf{FLTS}$, $L_r = \bigcup_{\mu \in L} L_{r(\mu)}$, where $L_{r(\mu)}$ is the label set of refinement transition system $r(\mu)$. We sometimes combine this notation with the τ and δ subscript notation, like in $L_{r\delta}$.

Definition 5.3.7 [Complete trace contraction]
Let $\sigma \in L^*_{r\delta}$, $r : L_\tau \to \mathbf{FLTS}$

$$\sigma\langle r\rangle_{rc} = \begin{cases} \{\epsilon\} & \text{if } \sigma = \epsilon \\ \{\lambda_1 \cdots \lambda_n \in L^*_\delta \mid \exists n > 0, \sigma_1, \ldots, \sigma_n \in L^*_{r\delta} : & \\ \sigma = \sigma_1 \cdots \sigma_n \wedge \forall 1 \leq i \leq n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))\} & \text{if } \sigma \neq \epsilon \end{cases}$$

Definition 5.3.8 [Incomplete trace contraction]
Let $\sigma \in L^*_{r\delta}, r : L_\tau \to \mathbf{FLTS}$.

$$\sigma\langle r\rangle_{inc} = \begin{cases} \emptyset & \text{if } \sigma = \epsilon \\ \{\lambda_1 \cdots \lambda_n \in L^*_\delta \,|\, \exists n > 0, \sigma_1, \ldots, \sigma_n \in L^*_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n & \\ \qquad \wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) & \\ \qquad \wedge \sigma_n \in \mathit{XStraces}(r(\lambda_n))\} & \text{if } \sigma \neq \epsilon \end{cases}$$

Definition 5.3.9 [Trace contraction] Let $\sigma \in L^*_{r\delta}$

$$\sigma\langle r\rangle = \sigma\langle r\rangle_{rc} \cup \sigma\langle r\rangle_{inc}$$

We extend the contraction definitions for sets of traces in the following way: $\Sigma\langle r\rangle_{rc} = \bigcup_{\sigma \in \Sigma} \sigma\langle r\rangle_{rc}$, $\Sigma\langle r\rangle_{inc} = \bigcup_{\sigma \in \Sigma} \sigma\langle r\rangle_{inc}$ and $\Sigma\langle r\rangle = \bigcup_{\sigma \in \Sigma} \sigma\langle r\rangle$ for $\Sigma \subseteq L^*_{r\delta}$.

Example 5.3.10 Trace contraction works similarly to trace refinement. We take the refined trace i€1·i€2·refund·o€2·o€1 from Example 5.3.6. We see that i€1·i€2 is in $\mathit{TXStraces}(r($i€3$))$, that refund is in $\mathit{TXStraces}(r($refund$))$ and that o€2·o€1 is in $\mathit{TXStraces}(r($o€3$))$. In our case this is relatively easy because there is no overlap in label sets: (i€1·i€2·refund·o€2·o€1)$\langle r\rangle_{rc}$ = {i€3·refund·o€3}. The incomplete contraction results in the empty set. □

Notice that due to our liberal constraints, the sets of complete and incomplete refinements may overlap (the same holds for the sets of contractions). This does not lead to problems. If a trace is only completely refined, we can relate it to the abstract system via complete contraction. If a trace is only incompletely refined, we can relate it to the abstract system by incomplete contraction. And if a trace is in the sets of complete and incomplete refinements, then we relate it to the abstract system by applying both complete and incomplete contraction. This may of course mean that there is more than one abstract trace. There is a pleasant relation between trace refinement and trace contraction, which states that trace contraction is the inverse of trace refinement and vice versa.

Proposition 5.3.11 Let $\sigma \in L_\delta^*, \sigma' \in L_{r\delta}^*, r : L_\tau \rightarrow \mathbf{FLTS}$

$$\sigma' \in \sigma[r] \Leftrightarrow \sigma \in \sigma'\langle r \rangle$$

$\square$

A nice relation between trace refinement and LTS refinement is that the refinement of suspension traces of the abstract system results in suspension traces of the refined system.

Theorem 5.3.12

$$Straces(s)[r] = Straces(s[r])$$

$\square$

The reason this theorem holds is a combination of definitions of trace refinement and LTS refinement, together with the constraints on the refinement function. The theorem would fail to hold if one of the constraints is not met.

5.4 ioco with refinement

In this section we introduce a new variant of the **ioco** implementation relation: $\mathbf{ioco}_r$. It expresses correctness of the concrete implementation with respect to the abstract specification and the refinement function. This relation is important, for **ioco** after refinement also gives a notion of correctness between the abstract system and the concrete implementation. We expect both notions to be pleasantly related and indeed we show that $\mathbf{ioco}_r$ and **ioco** are equivalent in the sense that they are equally powerful in discriminating implementations. This is a nice result, because it provides a sanity check on the action refinement definitions. On the other hand the result is not as nice as we hoped for, because the definition of $\mathbf{ioco}_r$ and our action refinement definitions are closely related. The main reason for this is the use of trace refinement and contraction in the definition of $\mathbf{ioco}_r$.

To give some intuition behind $\mathbf{ioco}_r$, it looks at the traces in terms of completely refined and incompletely refined traces. For completely refined traces it looks at the output behavior of the abstract system and of the refinement transition systems. For incompletely refined traces, it looks at the output behavior inside the refinement transition systems (incompletely refined traces end inside a refinement transition system).

The following defines the set of output actions that are allowed after a *completely* refined trace. Completely refined traces (at least) end in the final state of a refinement transition system. We use the abstract specification to find out which refinement transition systems to take into account.

Definition 5.4.1 Let $\sigma \in L^*_{r\delta}, s \in \mathbf{LTS}(I,U), r : L_\tau \to \mathbf{FLTS}$. We use the help set $\Sigma = (\sigma\langle r\rangle_{rc} \cap \mathit{Straces}(s))$

$$\begin{aligned} \mathit{out}_{rc}(s,\sigma,r) = \ & \bigcup_{\sigma'\in\Sigma,\mu\in \mathit{out}(s\ \mathbf{after}\ \sigma')\backslash\{\delta\}} \mathit{out}(r(\mu))\backslash\{\delta\} \\ & \cup\ (\bigcup_{\sigma'\in\Sigma} \mathit{out}(s\ \mathbf{after}\ \sigma') \cap \{\delta\}) \end{aligned}$$

The definition is straightforward, though rather technical, therefore we explain it with the following example.

Example 5.4.2 First we explain the first part of the formula which concerns the non-quiescent case: $\mathit{out}(r(\mu))\backslash\{\delta\}$ forall $\mu \in \mathit{out}(s\ \mathbf{after}\ \sigma')\backslash\{\delta\}\}$. Suppose that we want to compute $\mathit{out}_{rc}(s,\sigma,r)$ for our video game example with $\sigma = \text{i€2·i€1·play}$. We first compute $\Sigma = (\sigma\langle r\rangle_{rc} \cap \mathit{Straces}(s)) = \{\text{i€3·play}\}$. Then we compute $\mathit{out}(s\ \mathbf{after}\ \text{i€3·play})\backslash\{\delta\} = \{\text{game}\}$. Next we compute the final step: $\mathit{out}(r(\text{game}))\backslash\{\delta\} = \{\text{game}\}$.

The second part of the formula deals with quiescence: $\mathit{out}(s\ \mathbf{after}\ \sigma') \cap \{\delta\}$. For our case we have $\mathit{out}(s\ \mathbf{after}\ \text{i€3·play}) \cap \{\delta\} = \{\text{game}\} \cap \{\delta\} = \emptyset$. Hence $\mathit{out}_{rc}(\text{i€2·i€1·play}, s, r) = \{\text{game}\}$. □

The following definition computes the outset for *incompletely* refined traces. A trace σ is split up in sub-traces, such that all sub-traces –except the last one– are in the set *TXStraces* of some abstract action. The last sub-trace should be in the set *XStraces* of some abstract action. This requirement expresses that the trace ends inside a refinement. The goal is to end up with the set of outputs that are allowed within refinements.

Definition 5.4.3 Let $\sigma \in L^*_{r\delta}\backslash\{\epsilon\}, s \in \mathbf{LTS}(I,U), r : L_\tau \to \mathbf{FLTS}$.

$$\begin{aligned} \mathit{out}_{inc}(s,\sigma,r) = \{x \in \mathit{out}((r(\lambda_n)\ \mathbf{after}\ \sigma_n)\backslash\{\text{final}\}) \mid \\ \exists n > 0, \sigma_1,\ldots,\sigma_{n-1} \in L^*_{r\delta}, \lambda_1,\ldots,\lambda_{n-1} \in L_\delta : \\ \sigma = \sigma_1 \cdots \sigma_n \wedge \lambda_1 \cdots \lambda_n \in \mathit{Straces}(s) \\ \wedge \forall 1 \le i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \\ \wedge\, \sigma_n \in \mathit{XStraces}(r(\lambda_n))\} \end{aligned}$$

Like the complete case, this is a straightforward though rather technical definition. We explain it in the next example.

Example 5.4.4 Suppose that we want to compute $\mathit{out}_{inc}(s,\sigma,r)$ for our video game example with $\sigma = \text{i€2·refund}$. In Definition 5.4.3 we use the definition of incomplete contraction to refer to the last label of the incomplete contraction of σ. In our case there is only one solution: $\text{i€2·refund} \in \mathit{XStraces}(r(\text{i€3}))$; in terms of our definition, this means that $\sigma_n = \text{i€2·refund}$ and $\lambda_n = \text{i€3}$. To finish our example we compute: $\mathit{out}(r(\lambda_n)\ \mathbf{after}\ \sigma_n) = \{\text{o€2}\}$, therefore $\mathit{out}_{inc}(s,\sigma,r) = \{\text{o€2}\}$. □

The definition of $\mathbf{ioco}_r$ takes the union of both sets. This means that we have the set of all possible *refined* output actions after a refined trace. Our goal is that this is exactly the same set as the refined specification prescribes.

Definition 5.4.5 [$\mathbf{ioco}_r$]
Let $s \in \mathbf{LTS}(I, U), i \in \mathbf{IOTS}(I_r, U_r), r : L_\tau \to \mathbf{FLTS}$

$$i\ \mathbf{ioco}_r\ s \Leftrightarrow \forall \sigma \in \mathit{Straces}(s)[r] : \mathit{out}(i\ \mathbf{after}\ \sigma) \subseteq \mathit{out}_{rc}(s, \sigma, r) \cup \mathit{out}_{inc}(s, \sigma, r)$$

One may wonder how the $\mathbf{ioco}_r$ definition works out for traces that are both complete and incomplete refinements. In this case we compute both the out_{rc} and out_{inc} sets. For being in both sets means that the particular trace may end within a refinement and at the end of a refinement.

Proposition 5.4.6 Let $s \in \mathbf{LTS}(I, U), \sigma \in L^*_{r\delta}, r : L_\tau \to \mathbf{FLTS}$

$$\mathit{out}(s[r]\ \mathbf{after}\ \sigma) = \mathit{out}_{rc}(s, \sigma, r) \cup \mathit{out}_{inc}(s, \sigma, r)$$

□

We obtain Theorem 5.4.7 when we combine Theorem 5.3.12 and Proposition 5.4.6. Theorem 5.3.12 enables us to move back and forth between the abstract and refined traces and Proposition 5.4.6 gives us the required results on the *outset* of the refined system.

Theorem 5.4.7 Let $i \in \mathbf{IOTS}(I, U), s \in \mathbf{LTS}(I, U), r : L_\tau \to \mathbf{FLTS}$.

$$i\ \mathbf{ioco}_r\ s \Leftrightarrow i\ \mathbf{ioco}\ s[r]$$

□

5.5 Test-case refinement

With test-case refinement, we want to obtain test-cases with the required level of detail in order to test the IUT. There are two ways to obtain a refined test-suite. One way is to refine the abstract specification and generate a test-suite from the refined specification. Another way is to directly refine the abstract test-suite into a refined test-suite. Our results on LTS refinement enable us to use the former approach. When we use Tretmans' test generation algorithm, we obtain a test-suite that is complete with respect to **ioco** and the refined specification. In this section we examine how to obtain a refined test-suite by directly refining an abstract test-suite.

Our test-case refinement approach works in a similar way as trace refinement. In short, we make for every transition in an abstract test-case (that does not lead to **pass** or **fail**, i.e., a final state) a mini-test-case. This mini-test-case tests the behavior defined by a refinement transition system. Next we combine all mini-tests together to form a test-case and we add missing observations where necessary.

5.5.1 Mini-test generation

A mini-test is a special test-case for a refinement transition system. It does not have fail states and only looks at the defined outputs of the refinement transition system in an observation step. It has a special state final that indicates the final state of the mini-test.

Definition 5.5.1 [mini-test] A mini-test $\langle Q, I, U, T, \mathsf{start}, \mathsf{final}, \mathsf{Pass}\rangle$ is an acyclic FLTS $\langle Q, I, U, T, \mathsf{start}, \mathsf{final}\rangle$ with the addition of $\mathsf{Pass} \subseteq Q$: a set of pass states.

Definition 5.5.2 [mini-test generation]
Let $s = \langle Q_s, I_s, U_s, T_s, \mathsf{start}_s, \mathsf{final}_s\rangle \in \mathbf{FLTS}(I_s, U_s)$ be a refinement transition system. S is a set of states that we initialize with $\{\mathsf{start}_s, *\}$, where $*$ as a special symbol to forbid the observation of δ as a first observation. A mini-test mt is obtained from S by a finite number of recursive applications of one of the following non-deterministic choices. We denote the entire set of generated mini-tests for an FLTS s as $MT(s)$.

1. $mt :=$ **pass**.
2. $mt := \checkmark$ if $\mathsf{final}_s \in S$. We apply this rule only once for the entire mini-test generation. As a result there is only one state marked with $\checkmark$ (final state).
3. $mt := a; t'$ where $a \in I$, t' is obtained by applying the algorithm for $S' = S\backslash\{*\}$ **after** a. This rule is only applicable if $S\backslash\{*\}$ **after** $a \neq \emptyset$.
4. $mt := \Sigma\{x; t_x\}$, where $x \in out(S\backslash\{\mathsf{final}\})\}$, t_x is obtained by applying the algorithm for $S' = (S\backslash\{\mathsf{final}\})$ **after** x. This rule is only applicable when $* \notin S$ and $out(S\backslash\{\mathsf{final}\}) \neq \emptyset$.
5. $mt := \Sigma\{y; t_y\}$, where $y \in out(S\backslash\{*, \mathsf{final}\})\backslash\{\delta\}$, t_y is obtained by applying the algorithm for $S' = S\backslash\{*\}$ **after** y. This rule is only applicable when $* \in S$ and $out(S\backslash\{*, \mathsf{final}\})\backslash\{\delta\} \neq \emptyset$.

Rule 1 and 2 stop the recursion in the mini-test generation algorithm. **pass** is the regular way and $\checkmark$ has a special meaning as it results in the final state of the mini-test. Via this state we connect the mini-test with the start state of another mini-test (there can be only one such state, as we will explain in Section 5.5.2). Via rule 3 we perform a stimulus a in the case that $S\backslash\{*\}$ **after** a is not the empty set. We update the set S with $S' = S\backslash\{*\}$ **after** a. We perform observations with rule 4 and 5. Rule 5 is for the first observation (when $* \in S$) and rule 4 for all other observations. When we are in the final state we do not perform an observation, hence the exclusion of final from S (there are no outgoing transitions in final and δ is

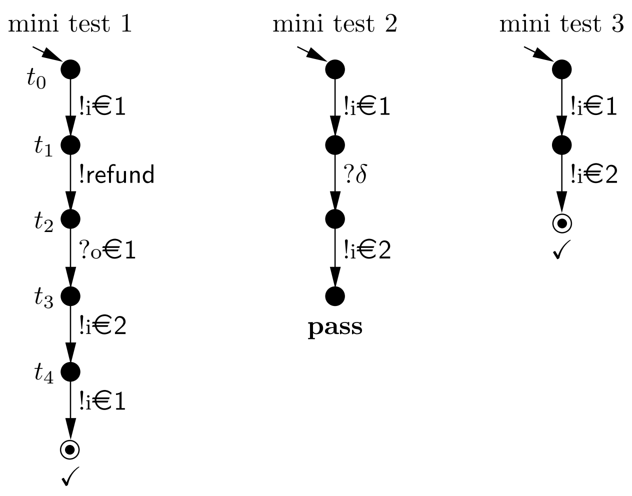


Figure 5.6: Example mini-test-case generation

not a valid observation in final). The constraints on the out sets in rules 4 and 5 prevent the unwanted result $mt = \Sigma\emptyset =$ **stop**

Just as δ observations are not allowed in final, δ observations are also not allowed in the start state of the refinement transition system. Or more precisely, they are not allowed as a first observation; if we return to the start state, for example via a loop, δ-observations are allowed. In order to prevent δ observations as the first observation of a mini-test, we use the symbol $*$. When $* \in S$, this means that it is the first observation of the mini-test, therefore δ is not a valid observation; hence the rule for $y; t_y$. When the outsets are empty it is possible to obtain $mt := \Sigma\emptyset$ (= **stop**), this is an unwanted situation, because we only want the mini-test to stop with rules 1 and 2. Therefore we check first if the outsets are empty.

Example 5.5.3 In Figure 5.6 we create some mini-tests for $r($i€3$)$ (Figure 5.3 on page 97 bottom right), to illustrate the mini-test generation algorithm. We show three mini-tests for this refinement transition system. We explain mini-test 1 in detail. We start with the state-set $S = \{s_0, *\}$. Suppose we non-deterministically choose the stimulus i€1 (step 3), which is possible, because S **after** i€1 $= \{s_3\}(\neq \emptyset)$. We obtain $mt =$ i€1$; t_0$. We continue with the "unrolling" of t_0 with state-set $S' = \{s_3\}$. Suppose we again choose a stimulus (step 3), in this case refund. We obtain $mt =$ i€1; refund$; t_1$. We continue with t_1 with state-set $S' = \{s_2\}$ **after** refund $= \{s_4\}$. Suppose we now choose an observation (step 4). The only defined output is o€1, resulting in $mt =$ i€1; refund; o€1$; t_2$ with the new state-set $S' = \{s_4\}$ **after** o€1 $= \{s_0\}$. This step is allowed because the outset is not empty. To obtain mini-test 1, we add two more stimuli: i€1 followed by i€2. In the same way we can generate the other mini-tests.

One final word on the observation of outputs (we use the same example). When we are in the start state, we are not allowed to observe quiescence. This is the reason why no observation is possible as the first test-action. When $S = \{s_0, *\}$, we obtain $out(S \backslash \{*, \mathsf{final}\}\ \mathbf{after}) \backslash \{\delta\} = \emptyset$. This results in $mt = \Sigma\emptyset$, which is forbidden because we want mini-test-cases to end with **pass** or $\checkmark$. □

To treat δ in a similar way as the other mini-tests, we introduce the set of mini-tests for the δ label: $MT(r(\delta))$.

Definition 5.5.4 [δ mini-test]
$MT(r(\delta))$ consists of one mini-test $\langle Q, I, U, T, \mathsf{start}, \mathsf{final}, \mathsf{Pass}\rangle$ with: $Q = \{\mathsf{start}, \checkmark\}$, $I = \emptyset$, $U = \{\delta\}, T = \{(\mathsf{start}, \delta, \checkmark)\}, \mathsf{final} = \checkmark, \mathsf{Pass} = \emptyset$.

5.5.2 Building the skeleton for refined test-cases

In this section we build skeletons for refined test-cases. We call them skeletons because they are not yet proper test-cases. For example, we still have to assign verdicts to some final nodes. We use a function to help us building the skeleton for refined test-cases: $f : Q \to L_\delta \to MT$. This is a function that takes an abstract state and an abstract label as input and delivers a mini-test for the abstract label. For the application of f on a state q we use the notation f_q rather than $f(q)$. To put it formally: let $q \in Q, \lambda \in L_\delta$ then $f_q(\lambda) \in MT(r(\lambda))$.

Definition 5.5.5 Let $t = \langle Q, I, U, T, \mathsf{start}, \mathsf{Pass}, \mathsf{Fail}\rangle \in \mathbf{TEST}(I, U)$ (abstract test-case) and $f : Q \to L_\delta \to MT$. We assume that the *states* of the mini-test-cases do not have labels in $L_{r\delta}$ and we assume their state sets to be disjoint. We define $t[f] = \langle Q_f, I_f, U_f, T_f, (\mathsf{start}, \checkmark), \mathsf{Pass}_f, \mathsf{Unknown}_f\rangle$ as follows:

$$
\begin{array}{ll}
Q_f & = Q_1 \cup \{(q, q') \mid q \in Q_1, q' \in U_{f\delta}\}, \text{ where} \\
 & \quad Q_1 = \{(q, q') \mid q \in Q, q' \in \bigcup_{\lambda \in L_\delta, q_1 \in Q} Q_{f_{q_1}(\lambda)}\} \\
I_f & = \bigcup_{\mu \in L} I_{r(\mu)} \\
U_f & = \bigcup_{\mu \in L} U_{r(\mu)} \\
T_f & = T_1 \cup T_2 \cup T_3, \text{ where} \\
 & \quad T_1 = \{((q_1, \checkmark), \mu, (q_1', q_2')) \mid q_1' \notin \mathsf{Fail}, \exists \mu' \in L_\delta : (q_1, \mu', q_1') \in T, \\
 & \hfill (\mathsf{start}_{f_{q_1'}(\mu')}, \mu, q_2') \in T_{f_{q_1'}(\mu')}\} \\
 & \quad T_2 = \{((q_1, q_2), \mu, (q_1, q_2')) \mid q_1 \notin \mathsf{Fail}, q_2 \neq \checkmark, \\
 & \hfill \exists q \in Q, \mu' \in L : (q, \mu', q_1) \in T, (q_2, \mu, q_2') \in T_{f_{q_1}(\mu')}\} \\
 & \quad T_3 = \{((q_1, q_2), \mu, ((q_1, q_2), \mu)) \mid ((q_1, q_2), \mu) \in Q_f, \\
 & \hfill \exists \mu' \in U_{r\delta}, q \in Q_1 : ((q_1, q_2), \mu', q) \in T_1 \cup T_2, \\
 & \hfill \nexists q' \in Q_1 : ((q_1, q_2), \mu, q') \in T_1 \cup T_2\} \\
\mathsf{Pass}_f & = \{(q, q') \mid q \in \mathsf{Pass}, q' = \checkmark\} \cup \{(q, q') \mid \exists \mu \in L_\delta, m \in MT(r(\mu)) : \\
 & \hfill q' \in \mathsf{Pass}_m\} \\
\mathsf{Unknown}_f & = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in U_{r\delta}\}
\end{array}
$$

The outcome of this construction is called a *test skeleton.* They have the following properties:

- Transitions in T_1 connect the end-state of one mini-test to the start-state of another.
- Transitions in T_2 add all the other transitions of mini-tests; other meaning not a start or end-state.
- Transitions in T_1 and T_2 do not guarantee that all possible actions in $U_{f\delta}$ are added to a test-case observation step (remember that a test-case does either one input action, or observes all possible outputs of the system under test). T_3 makes sure that any missing output observation is added.

The definition is rather technical and we will explain it in some more detail. The creation of the test skeleton is quite similar to Definition 5.2.1 (LTS refinement). The main differences are the transitions in T_3 and the Pass and Unknown states (we use **unknown** to refer to states in Unknown, analogous to **pass** and **fail**). T_3 adds all undefined observations: mini-tests only generate observations for defined output actions. Pass states are state tuples where the first state is a pass state from the abstract test-case and the second state is $\checkmark$, or where the second state is a pass state from the mini-test. Unknown states are all the observations that we add in T_3. The Unknown states are tuples where the first element is a state tuple in Q_1 and the second element is a label (the label of the incoming transition). In this way we can uniquely identify the Unknown states. We use this information later on to find out if the these states can be considered to be **pass** or **fail** states.

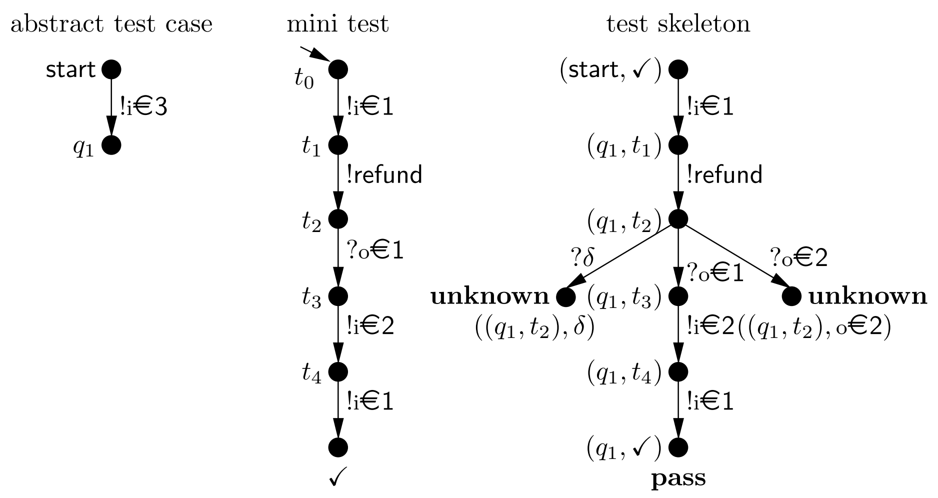


Figure 5.7: Skeleton building example

The reason that we use Unknown states is that a single test-case does not have enough information to assign failures to the refined test-case. The mini-tests and the refinement transitions systems only provide information to determine if a trace is a suspension trace of the refined system. It does not provide enough information to determine that a trace is *not* a suspension trace. We illustrate this in the following example.

Example 5.5.6 Figure 5.7 depicts the way to come from an abstract test-case and mini-test to a test skeleton. We take only one transition of the abstract test-case into account to keep the example compact. In the Figure we see an abstract test-case (left), a mini-test (middle, we re-use the mini-test from Example 5.5.3) and a test skeleton (right). We assume that $f_{q_1}(\text{i€3})$ has mini-test 1 as its result (we refer to mini-test 1 as m_1). We start with the start state $(\text{start}, ✓)$ and the abstract transition $(\text{start}, \text{i€3}, q_1)$, with $q_1 \notin$ Fail. We see that $(t_0, \text{i€1}, t_1) \in T_{m_1}$, via T_1 we add $((\text{start}, ✓), \text{i€1}, (q_1, t_1))$ to the transition set of the test skeleton. For the state (q_1, t_1) we see that $q_1 \notin$ Fail, $t_2 \neq ✓$, $(\text{start}, \text{i€3}, q_1) \in T$ and $(t_1, !\text{refund}, t_2) \in T_{f_{q_1}(\text{i€3})}$, therefore T_2 adds $((q_1, t_1), !\text{refund}, (q_1, t_2))$ to the test skeleton. The ?o€1, !i€2 and !i€1 transitions work in the same way (T_2). This leaves us with the T_3 transitions, we illustrate this with the observation $((q_1, t_2), ?\text{o€1}, (q_1, t_3))$. T_3 tells us that for every missing output we add a transition. Suppose that the output set is $\{\text{o€1}, \text{o€2}, \delta\}$, this means that the o€2 and δ observations are missing, therefore we add: $((q_1, t_2), \delta, ((q_1, t_2), \delta))$ and $((q_1, t_2), ?\text{o€2}, ((q_1, t_2), \text{o€2}))$.

Note that at this point it is not possible to tell if these two unknown states should be pass or fail states, based on the information of the mini-tests and the refinement function. Suppose that based on the refinement transitions system $r(\text{i€3})$ we conclude that the observation o€2 is not allowed

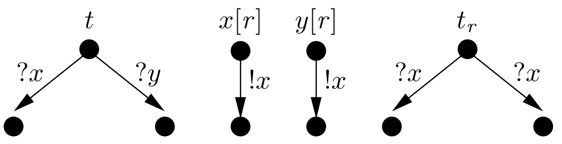


Figure 5.8: Example of a non deterministic test-skeleton

and that is should lead to fail. There is no restriction in our refinement approach that prevents this behavior from occurring in another part of the refined system. Just imagine an abstract action a (enabled at the same time that i€3 is enabled) which refinement transition system enables the trace i€1·refund·o€2. The other way around, we cannot just lead the observation of o€2 to pass. Suppose that o€2 is only defined in r(i€3), that means that the observation of o€2 will never lead to **fail** and that we can never detect this failure. □

5.5.3 Turning test-case skeletons into proper test-cases

The skeletons of refined test-cases as defined above have two problems. First they have **unknown** states and no fail states. Second they are not proper test-cases. It can be the case that an observation has two different transitions for the same output action, therefore it may violate Definition 2.5.1 (test-case).

The non-determinism in skeletons arises because we allow refinement transition systems to have overlapping label sets. We depict the problem in Figure 5.8. We see an abstract test-case (left), refinement transition systems for x and y (middle) and the skeleton (right). The abstract test-case has the observations x and y. The refinement function tells us that x is refined to itself and that y is refined to x. This results in the non-deterministic skeleton on the right. In principle, the non-deterministic observations are not a big problem. A refined test-case with this behavior can still be executed against an implementation. However, it violates the commonly used definition of test-cases and therefore we determinize the skeletons first. We do not go into the details of determinization here, we assume the skeleton to be deterministic. Determinization is a well-studied subject, for our means the so called λ-closure, also known as powerset or subset construction works fine (see for example [Sud88], page 151-153). An alternative may be the splitting of skeletons whenever a non-deterministic observation occurs. We did not investigate if this is a better option than other determinization techniques.

We introduced the **unknown** states because sometimes we do not know what verdict to give to an observation. This arises because of the few restrictions we place on the action refinement function. The result is that a refined trace can have more than one abstract trace as its original. As a result it may be the case that we cannot base our verdict only on the refinement function. By using the information in the refinement function and the abstract test-suite we may resolve **unknown** states into **pass** and **fail** states. It may be the case that there are states that we cannot resolve and for these states we introduce the *inconclusive* verdict. Below we define *extended* test-cases that take inconclusive verdicts into account. Analogous to **pass** and **fail** we use **inconclusive** to refer to states in Inconclusive.

Definition 5.5.7 [Extended test-case] An extended test-case $t = \langle Q, S, R, T, \mathsf{start}, \mathsf{Pass}, \mathsf{Fail}, \mathsf{Inconclusive}\rangle$ is a test-case $\langle Q, S, R, T, \mathsf{start}, \mathsf{Pass}, \mathsf{Fail}\rangle$ with the addition of Inconclusive: a set of inconclusive states such that $\mathsf{Pass} \cap \mathsf{Inconclusive} = \mathsf{Fail} \cap \mathsf{Inconclusive} = \emptyset$.

The final test-case refinement step is the removal of the Unknown set. We change the **unknown** verdict to **pass**, **fail** or **inconclusive** in the following way.

Definition 5.5.8 [Verdict assignment] Let AT be a set of abstract test-cases. Let $t \in AT$ and $s = \langle Q, S, R, T, \mathsf{start}, \mathsf{Pass}, \mathsf{Unknown}\rangle$ a test skeleton derived from t as described in Definition 5.5.11. We transform s to an extended test-case, denoted by $\langle Q, S, R, T, \mathsf{start}, \mathsf{Epass}, \mathsf{Efail}, \mathsf{Inconclusive}\rangle$. Epass, Efail and Inconclusive have the following definition:

$$\begin{array}{ll}
\mathsf{Epass} & = \mathsf{Pass} \cup \{q \in \mathsf{Unknown} \mid \exists \sigma \in L^*_{r\delta} : \mathsf{start} \xrightarrow{\sigma} q \\
& \qquad\qquad \wedge \exists \sigma' \in \sigma\langle r\rangle, t \in AT, q' \in Q_t \backslash \mathsf{Fail}_t : t \xrightarrow{\sigma'} q'\} \\
\mathsf{Efail} & = \{q \in \mathsf{Unknown} \mid \exists \sigma \in L^*_{r\delta} : \mathsf{start} \xrightarrow{\sigma} q \\
& \qquad\qquad \wedge \forall \sigma' \in \sigma\langle r\rangle, \exists t \in AT, q' \in \mathsf{Fail}_t : t \xrightarrow{\sigma'} q'\} \\
\mathsf{Inconclusive} & = \mathsf{Unknown} \backslash (\mathsf{Epass} \cup \mathsf{Efail})
\end{array}$$

In words, rule 1 states that if there is a contraction of σ for which there is an abstract test-case that leads to a non-fail state, then we turn q into a pass state. Rule 2 states if for all contractions of σ there is an abstract test-case leading to fail (for the abstract trace) then we turn q into a fail state. This includes the case that $\sigma\langle r\rangle = \emptyset$. In all other cases we move q to Inconclusive. The intuition behind Epass is as follows. Suppose for a trace that leads to an unknown state we have an abstract test-case for its contraction that does not lead to fail. This means that we can refine that abstract test-case into a test-case that can perform its refinements, including the trace leading to the unknown state. In other words this is a trace that the implementation should pass. For the Efail case on the other hand, if all contractions of the trace leading to unknown have a test-case leading to fail, we know that this refinement is impossible and should therefore fail. We call the set of remaining unknowns Inconclusive.

An interesting question is if we can do better than this, i.e., are there states in Inconclusive that can become **pass** or **fail** states. In the case of **pass** it is easy to see that this is not the case (with the aid of the fail-fast

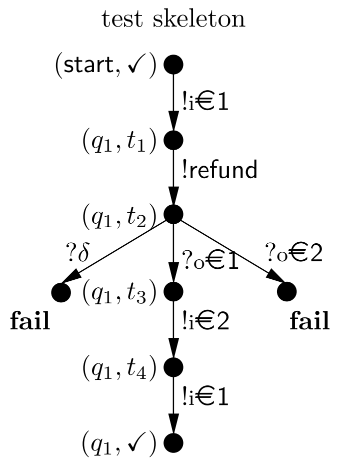


Figure 5.9: Verdict assignment

property that we will introduce in the next section). If there is a test-case for a contraction that leads to a non fail state, we know (fail-fast) that the contraction is a suspension trace of the abstract system and as a result the refined trace is a trace of the refined system (Theorem 5.3.12). In the case of the **fail** states it is not clear if we can do better and more research is needed. It is however clear that if none of the contractions is a suspension trace of the abstract system, then the refined trace is also not a suspension trace of the refined system. Note that for the second case it is important that *all* contractions should have an abstract test-case leading to fail. If there is only one abstract trace and one test-case leading to **pass** (rule 1), we know that the refined trace is a suspension trace of the refined system.

In general it seems a good idea to take the Pass and Inconclusive sets together. Inconclusive means that we do not know if the trace is a suspension trace of the refined system. Interpreting inconclusive as failure may introduce unsound test-cases and therefore seems a bad idea. Interpreting inconclusive as pass does not have this effect. We show in the next section that under some circumstances we can interpret the Inconclusive as failures. This is the reason that we identify the **inconclusive** as a separate verdict. Unless stated otherwise, the inconclusive verdict is interpreted as the pass verdict, turning the extended test-case in a normal test-case.

Example 5.5.9 In Example 5.5.6 we built a test skeleton with the traces $!\mathsf{i€1}\cdot!\mathsf{refund}\cdot?\delta$ and $!\mathsf{i€1}\cdot!\mathsf{refund}\cdot?\mathsf{o€2}$ leading to **unknown** states. It depends on the refinement function and the abstract test-suite if these traces should lead to **pass**, **fail** or **inconclusive**. When we use the refinement function of Figure 5.3 on page 97 we get the contractions: $(!\mathsf{i€1}\cdot!\mathsf{refund}\cdot?\delta)\langle r\rangle = \emptyset$ and $(!\mathsf{i€1}\cdot!\mathsf{refund}\cdot?\mathsf{o€2})\langle r\rangle = \emptyset$, therefore the states become fail states. The result is depicted in Figure 5.9. □

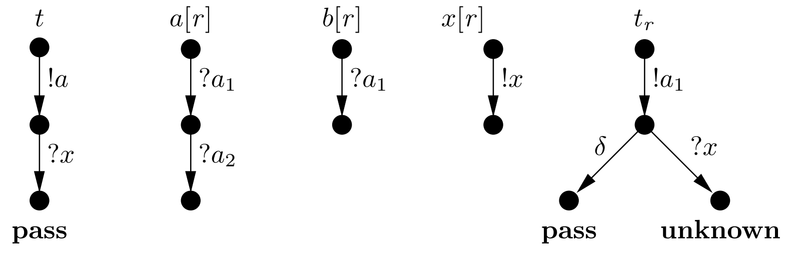


Figure 5.10: Verdict assignment in more detail

In the following example we treat the verdict assignment definition in more detail and show why we need the inconclusive verdict.

Example 5.5.10 Suppose we have an abstract test-set with one test-case: a·x·**pass** (left-hand site in Figure 5.10). We have the refinement transition systems $a[r]$, $b[r]$ and $x[r]$ also in the figure. a is refined into input actions a_1 followed by a_2, b by input action a_1 and x by itself. The test-skeleton on the right-hand site of the figure is generated from the abstract test-case. We want to transform the **unknown** state into **pass**, **fail** or **inconclusive**. A valid contraction of a1·x is b·x. This trace is not covered by our abstract test-set, hence we do not know if it is an allowed suspension trace of the abstract system. This means that the **unknown** state becomes and **inconclusive** state after verdict assignment. If we would have had an abstract test-case that covered the trace b·x, the state would have become a **pass** state. In case we would have had an abstract test-case leading to **fail** after b·x the state would have become a **fail** state. □

From now on we use the term test-case refinement to refer to the end result of the test-case refinement steps (mini-test generation, skeleton building and verdict assignment). This means that for an abstract test-case t, $t[r]$ denotes the set of refined extended test-cases. We extend test-case refinement to test-suites in a straightforward manner. Let $T \subseteq \mathbf{TEST}(S, R)$ be an abstract test-suite then, $T[r] = \bigcup_{t \in T} t[r]$. In order to get all possible combinations of mini-tests in the refined test-cases we use all possible functions f.

Definition 5.5.11 Let $t = \langle Q, I, U, T, \mathsf{start}, \mathsf{Pass}, \mathsf{Fail} \rangle \in \mathbf{TEST}(I, U)$.

$$t[r] = \{t[f] \mid f : Q \to L_\delta \to MT, \text{ with } \forall q \in Q, \lambda \in L_\delta : f_q(\lambda) \in MT(r(\lambda))\}$$

5.5.4 Completeness of test-case refinement

In this section we show under which circumstances the refinement of a complete abstract test-suite results in a complete refined test-suite. The completeness property consists of two parts: soundness and exhaustiveness. First we introduce the property fail-fast.

We assume that test-cases give a fail verdict as soon as a response occurs that violates **ioco**. We call this property *fail-fast*. The definition below expresses that when we remove the last action of a trace (to test system s) leading to a fail state, we should obtain a suspension trace of s. Together with soundness this is a helpful property, because we know that a test-case that performs the prefix of the trace leading to fail, does not lead to a fail state. Furthermore, we know that a trace not leading to a fail state is a suspension trace. As mentioned in the previous section, fail-fast is important for Definition 5.5.8 (verdict assignment), especially for the definition of Epass.

Definition 5.5.12 [Fail-fast]
Let $t \in \mathbf{TEST}(I, U)$ be a test-case to test system s and $x \in L_\delta$. We call t fail-fast for s if the following holds:

$$t \xrightarrow{\sigma} q \wedge q \notin \mathsf{Fail} \Rightarrow \sigma \in \mathit{Straces}(s)$$

Proposition 5.5.13 Let $T \subseteq \mathbf{TEST}(I, U)$ be sound and fail-fast, $r : L_\tau \to \mathbf{FLTS}, t_r \in T[r], \sigma \in L^*_{r\delta}$.

$$t_r \xrightarrow{\sigma}_r \mathbf{fail} \Rightarrow \exists \sigma' \in \mathit{Straces}(s[r]), x \in U_{r\delta} : \sigma = \sigma' {\cdot} x \wedge x \notin \mathit{out}(s[r] \ \mathbf{after}\ \sigma')$$

□

With this proposition it is straightforward to prove that a sound and fail-fast test-suite is sound again after refinement.

Theorem 5.5.14 [Soundness of the refined test-suite]
Let $t \in \mathbf{TEST}(I, U), s \in \mathbf{LTS}(I, U)$ and let t be fail-fast.
(t is **sound** w.r.t. **ioco** and s) $\Rightarrow$ ($t[r]$ is **sound** w.r.t. **ioco** and $s[r]$) □

To prove that an exhaustive abstract test-suite is again exhaustive after refinement, we need an extra constraint: *conformance trace completeness.* A test-suite is *conformance trace complete* with respect to a transition system s if for every suspension trace of s there is a test-case to perform this trace. We introduce this property, because a complete test-suite is not necessarily conformance trace complete. As discussed in [vdBRT05], a complete test-suite can be optimized by removing test-cases that are superfluous (for example because they always lead to pass). Thus it might be the case that we have a test-suite that is not conformance trace complete with respect to its specification. This means that we may have an abstract trace σ for which we do not have a test-case. Suppose that there is an error in the implementation for some of the traces in $\sigma[r]$. This means that we cannot construct a refined test-case to perform these refined traces, so we cannot determine the incorrectness. For the exhaustiveness results of this section to hold, we need test-cases for these traces in the test-suite (in order to refine them).

Definition 5.5.15 [Conformance trace completeness]
Let $s \in \mathbf{LTS}(I, U)$, $T \subseteq \mathbf{TEST}(I, U)$. T is conformance trace complete with respect to s if:

$$\forall \sigma \in \mathit{Straces}(s) : \exists t \in T : t \xrightarrow{\sigma}$$

When the abstract test-suite is conformance trace complete, the refined test-suite will give the verdict **inconclusive** or **fail** for non-conforming observations.

Proposition 5.5.16 Let $s \in \mathbf{LTS}(I, U), \sigma \in \mathit{Straces}(s[r])$ and let T be a conformance trace complete, exhaustive and fail-fast test-suite for s with respect to **ioco**.

$$x \notin \mathit{out}(s[r] \textbf{ after } \sigma) \Rightarrow \exists t_r \in T[r], q \in (\mathsf{Fail}_{t_r} \cup \mathsf{Inconclusive}_{t_r}) : t_r \xrightarrow{\sigma \cdot x}_r q$$

□

When an abstract test-suite T is fail-fast and conformance trace complete, we have a special situation. In this case we can interpret the inconclusive states as fail states. The rationale behind this is that, because there is an (abstract) test-case for every suspension trace, we can also construct test-cases for the all the refined traces. When we have a refined test-case leading to **inconclusive** this means that there simply was no abstract test-case to begin with and therefore the trace is not a suspension trace.

Proposition 5.5.17 Let T be complete, fail-fast and conformance trace complete with respect to s and **ioco** and let $t_r \in T[r]$.

$$t_r \xrightarrow{\sigma \cdot x} \textbf{inconclusive} \Rightarrow x \notin \mathit{out}(s[r] \textbf{ after } \sigma)$$

□

For the following theorem and corollary we take the union of $\mathsf{Inconclusive}$ and Fail as the set of fail-states. The result is a test-case in terms of Definition 2.5.1 (test-case).

Theorem 5.5.18 [Exhaustiveness of the refined test-suite]
Let $s \in \mathbf{LTS}(I, U), T \subseteq \mathbf{TEST}(I, U)$ be a conformance trace complete and fail-fast test-suite.

$$\begin{aligned} &T \text{ is } \textbf{exhaustive} \text{ w.r.t. } \textbf{ioco} \text{ and } s \\ &\qquad \Rightarrow T[r] \text{ is } \textbf{exhaustive} \text{ w.r.t. } \textbf{ioco} \text{ and } s[r] \end{aligned}$$

□

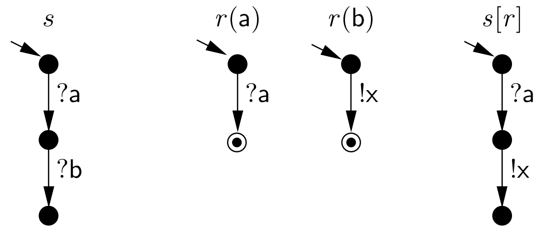


Figure 5.11: Example for delta preservation

Proposition 5.5.16 and the fact that we can treat **inconclusive** states as **fail** are the main ingredients in the proof of this theorem.

Tretmans' test-case generation algorithm generates a complete, conformance trace complete and fail-fast test-suite. Hence refinement of such a test-suite gives us a complete test-suite with respect to $\mathbf{ioco}_r$ and the abstract specification.

Corollary 5.5.19 The refinement of a complete test-suite generated with Tretmans' algorithm for test-case generation, is complete with respect to $\mathbf{ioco}_r$ and the abstract specification.

5.6 Constraints revisited

Before we conclude this chapter, we want to revisit the constraints on our action refinement function. We now have all the information to explain what can go wrong if one of the constraints is violated.

The first three constraints in Section 4.6 – refinements of input actions are only allowed to start with an input action, refinements of output actions are only allowed to start with an output action and τ is refined to τ – are to make sure that our refinements are δ-preserving and δ-reflecting. With δ-preserving we mean that we preserve (i.e., do not remove) quiescence that originates from the abstract system. With δ-reflecting we mean that we do not add quiescence in the refined system/test-case at places where there is none in the abstract case. Without theses constraints we run into problems with trace and test-case refinement. We illustrate this in the following example.

Example 5.6.1 In Figure 5.11 we see on the left-hand side an abstract specification s. Its functionality is simple: a followed by b. This means that the trace a$\cdot\delta\cdot$b is a suspension trace of s. We refine a to itself and we violate constraint 2 by refining b to the output action x. In the refined system ($s[r]$ on the right-hand side) δ is not allowed anymore after a: δ is not preserved after refinement. How can we refine the trace a$\cdot\delta\cdot$b with these refinements? The only way to know if δ is allowed in the refined system, based on only the abstract trace and the refinement systems seems impossible. In order to identify where quiescence is allowed we would need information about the abstract system.

The other way around we can introduce quiescence in the refined system at places where there was none in the abstract system. In Figure 5.12 we show a similar refinement as before, but now we refine the output action x to the input action b. We see that a·x is a trace of the abstract system (δ is not allowed after a). In the refined system on the right-hand side, we see that a·δ·b is a trace of the refined system; δ is now allowed after a. This means that based on an abstract trace and the refinement relation alone, we cannot compute all refined traces of the abstract trace anymore. Like in the delta preservation case, we need the abstract system. □

As the example shows, we need constraints 1, 2 and 3 to handle δ-actions in trace refinement. We could of course ignore δ-actions, altogether. This enables us to refine traces, albeit that the refined traces will lack δ-actions. The result is that Theorem 5.3.12 would not hold anymore, because refinement of the suspension traces of the abstract system will be at most a proper subset of the suspension traces of the refined system. Similarly we will not be able to prove completeness of refined test-cases Theorem 5.5.14 and Theorem 5.5.18.

Restrictions 3 and 4 – τ is refined to τ and no forgetful refinement – are to prevent curious refinements from happening. One of them is δ-reflection as we discussed above. More general, when we allow invisible actions to become visible in the refinement or, vice versa, visible actions to become invisible, it is possible to create bizarre refinements. For example a refinement in which we make all behavior invisible. In terms of our classification in Section 4.5 these constraints are related to observability. When an action is refined to invisible behavior, or vice versa, we may not be able anymore to relate the abstract and the refined behavior. Removing restrictions 3 and 4 is not so much a problem for transition system refinement, it is however for trace and test-case refinement. As with constraints 1, 2 and 3, removing restrictions 3 and 4 have the result that Theorem 5.3.12 does not hold anymore for traces and that Theorem 5.5.14 (soundness) and Theorem 5.5.18 (exhaustiveness)

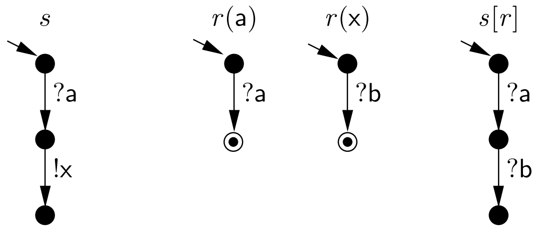


Figure 5.12: Example for delta reflection

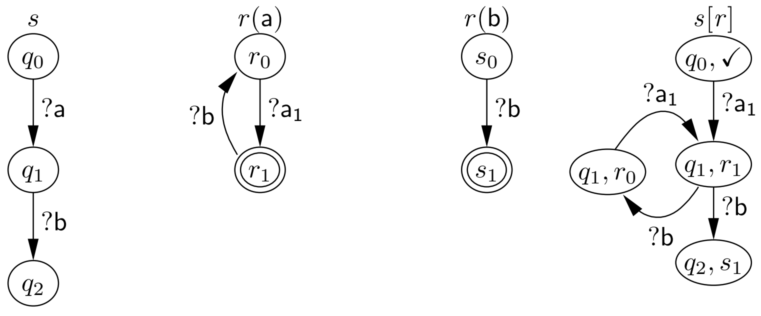


Figure 5.13: Observability constraint example

do not hold anymore for test-cases.

Note that forgetful refinement is actually implied by the constraint 1, 2 and 3, together with the constraint that the start-state of a refinement should be different from the end-state. It seems that in the future we can drop it altogether.

Restriction 5 – no outgoing transitions in the final state – makes it possible to determine where the refinement behavior ends in the refined system. This requirement is related to part 1 of the congruence requirement of Section 4.4. We need a clear termination point of refined behavior. We choose to do this via the final state. However, the final state is not a termination point of the refined behavior when outgoing transitions are possible. Without outgoing transitions we are sure that the final state signals the end of the behavior of the refinement transition system. We use this information in the proof of Theorem 5.3.12. We illustrate the constraint in the following example.

Example 5.6.2 We reuse the figure from the observability constraint (see Figure 5.13). We see an abstract system on the left-hand side that can per-

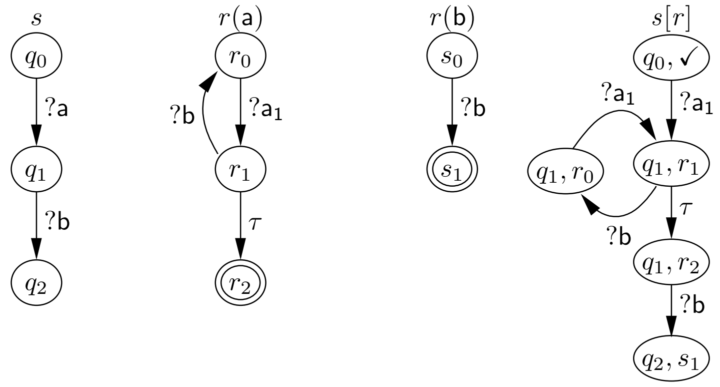


Figure 5.14: Observability constraint example, part two

form a followed by b. We refine b to itself and a to a_1, followed by zero or more times (b followed by a_1), thus violating constraint 5. $s[r]$ on the right-hand side is the resulting refined system. The problem at hand is that in the refined system we need to be able to tell when we reached the end of refinement behavior. We need this in order to relate the behavior in the refined system to behavior in the abstract system. In terms of Figure 5.13 on the preceding page this means that we want to know which of the executions $(q_0, \checkmark)\mathsf{a}_1(q_1, r_1)\mathsf{b}(q_1, r_0)$ and $(q_0, \checkmark)\mathsf{a}_1(q_1, r_1)\mathsf{b}(q_2, s_1)$ are related to execution $q_0\mathsf{a}q_1\mathsf{b}q_2$ of the abstract system s. In our proofs we use the second element of a state pair for this. If the second element of a state pair is the final state of a refinement transition system, we interpret this as the end of the refined behavior of the refinement transition system (to which the end state belongs). In our figure this means that (q_1, r_1) signals the end of the refined behavior of $r(\mathsf{a})$. When end states are allowed to have outgoing transitions this interpretation does not hold, because the transition $(q_1, r_1) \xrightarrow{\mathsf{b}} (q_1, r_0)$ is part of $r(\mathsf{a})$.

In Figure 5.14 on the facing page we model the same behavior without outgoing transitions in the final state of $r(a)$. As one can see in $s[r]$, we can clearly distinguish refinements from a that have reached the final state, from refinements that are not yet finished: (q_1, r_2) signals the end of refined behavior from $r(\mathsf{a})$. □

The observant reader may have noticed that in the future we can remove constraint 5. The important part is that we know when refined behavior has ended successfully (remember the congruence requirement from Section 4.4). We know this already from the context of the refinement transition system, because of the final state of the refinement transition system. The next thing to know is if an outgoing transition, of a state-pair with a final state as its second state is related to abstract or refined behavior. It seems that we get this information from Definition 5.2.1 (LTS refinement). Look again at $s[r]$ in Figure 5.13 on the preceding page. We can distinguish between behavior originating within a refinement transition system (before the final state) and behavior originating from another refinement system (after the final state) by looking at the first state of the state pairs in the refined system. In the figure $(q_1, r_1) \xrightarrow{\mathsf{b}} (q_1, r_0)$ can be distinguished from $(q_1, r_1) \xrightarrow{\mathsf{b}} (q_2, s_1)$ because the first state in the state pair changes from q_1 to q_2, indicating that the refined behavior of $r(\mathsf{a})$ has finished.

Note that contrary to the previous constraints, constraint 5 is important for the refinement of transition systems. For trace refinement it is not problematic if a final state has outgoing transitions. On a suspension trace level there is no difference between the refined systems in Figure 5.13 on the facing page and Figure 5.14 on the preceding page. The trace a_1 and $\mathsf{a}_1{\cdot}\mathsf{b}$ are refined traces of a and $\mathsf{a}_1{\cdot}\mathsf{b}$ is a refined trace as well of $\mathsf{a}{\cdot}\mathsf{b}$. Vice versa, we can contract $\mathsf{a}_1{\cdot}\mathsf{b}$ to a and $\mathsf{a}{\cdot}\mathsf{b}$.

5.7 Conclusion

In this chapter we addressed action refinement in model-based testing: the problem that a model, or a test-case, does not have enough information to properly test the IUT, because some actions are unknown to the IUT, or because the model, or test-case, is incorrect because of wrong abstractions. We presented action refinement as a way to add information to models and test-cases by replacing abstract actions with more complex behavior. We showed how to use action refinement to refine traces, transition systems and test-cases for a special case of action refinement: atomic action refinement. We also introduced a new implementation relation $\mathbf{ioco}_r$ that relates an abstract model to a concrete implementation.

It is important to realize that atomic and non-atomic action refinement both have their merits. In our opinion the choice between these two depends on the application at hand. In this chapter we showed the following main results:

- The set of refined traces of the abstract system is equal to the traces of the refined system. This is important because it means that trace refinement is compatible with LTS refinement; we do not leave some traces out.
- The refinement of a complete abstract test suite that is conformance trace complete and fail fast, results in a complete refined test suite. This result is important because it shows that directly refining a test suite is equivalent with first refining a system and then generating a test suite.

With the knowledge we have now, it seems that we can reduce the number of constraints on the refinement function. The only ones that seem necessary are the constraints on delta-preservation and reflection (1 to 3). Furthermore we need our refinement transition system to have an explicit end state.

Our refinement definitions, especially the ones for LTS-refinement and test-case refinement are rather technical. It would be beneficial for the developed theory to make the definitions easier to read and understand. We think that lifting the definitions to a process notation will improve the readability.

At first it was surprising how hard test-case refinement turned out to be. The reason is that test-cases have only partial knowledge of the specification. They literally are a set of traces: traces ending in a non-fail state are suspension traces of the specification and traces ending in fail state are not. The suspension traces can be refined in a way similar to trace refinement (with the use of mini-tests). However the difficulty lies in figuring out which traces do not belong to the set of suspension traces of the refined system.

The abstract test-case and the refinement function alone provide not enough information to do this.

It would be interesting to generalize the theory to non-atomic refinement. The literature shows some solutions for non-atomic refinement of specifications. Most of them are in the realm of truly concurrent behavior models, or process algebras (without distinction between input and output actions). This is unfortunate, because our test theory is based on labeled transition systems theory with interleaving semantics. Non-atomic trace and test-case refinement will be a challenge. In our atomic case we were able to use constraints to enable refinement of traces and test-cases without knowledge of the abstract system. For non-atomic action refinement it is not clear if this is still possible. Seeing the difficulty of identifying fail states in refined test-cases for atomic action refinement (and the strong dependability on the abstract test-suite), we believe that non-atomic test-case refinement will be quite a challenge.

5.8 Directions for further research

We end this section with some gedanken experiments on *non-atomic refinement* and *relaxed refinement*. We did not prove any properties on theories in this section, so it is not clear if the sketched approach might work. However we do think that is is an interesting direction in action-refinement research based on LTSs. Furthermore, it sketches some interesting action refinement scenarios and dilemmas.

5.8.1 Non-atomic model refinement

In Section 4.4 we showed the limitations of LTS models with interleaving semantics for non-atomic action refinement. The main reason for this is that with interleaving semantics we lose information about the independence between actions, for example independence that is the result of parallelism. There are examples of process algebraic and event structure approaches for non-atomic action refinement [GR01]. For action refinement in model-based testing, this leaves us with a problem, because the model-based testing theory (at least the **ioco** theory) is based on LTSs. A possibility is to adapt our test theory to event structures or to base it directly on a process algebra. Another possibility is to use models that capture the notion of independence and are compatible with LTS based models. An example of such a model is the Asynchronous Labeled Transition System (ALTS) [Bed88]. A more specific flavor of ALTSs, called Labeled Transition Systems with Independence (LTSI) was introduced by Winskel ([WN94]). We will go into a little detail to explain how action refinement with LTSI might work.

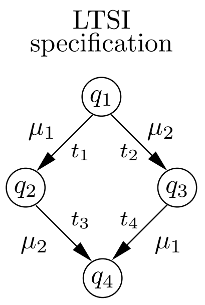


Figure 5.15: LTSI model of Example 5.8.2

Definition 5.8.1 [Transitions System with independence] A transition system with independence is a 6-tuple $\langle Q, I, U, T, \mathsf{start}, R\rangle$ where $R \subseteq T \times T$ is an irreflexive, symmetric relation and $\langle Q, I, U, T, \mathsf{start}\rangle$ is a transition system.

With $\mathbf{LTSI}(I, U)$ we denote the class of LTSI with input alphabet I and output alphabet U. As we can see an LTSI is basically an LTS. On top of this it has a relation R that keeps track of the independencies between transitions. When we have $t_1 R t_2$ for two transitions t_1 and t_2 this means that t_1 and t_2 are *independent* of each other, i.e., the order of appearance is flexible. In case $\neg(t_1 R t_2)$, transition t_1 is dependent of transition t_2; this means that the order of appearance of the actions of t_1 and t_2 is fixed and t_1 can not occur before t_2. We use the notation R_d to indicate the inverse of independence: *dependence*. We illustrate how LTSIs can help us to keep track of concurrent behavior in the following example.

Example 5.8.2 In Figure 5.15 we show a transition system that either models $(\mu_1 \parallel \mu_2)$ or $(\mu_1; \mu_2 + \mu_2; \mu_1)$. Based on the dependency relation between the transitions an LTSI can distinguish the parallel case from the choice case. If $t_1 R t_2$, $t_1 R t_3$, $t_2 R t_4$ and $t_3 R t_4$, we know that μ_1 and μ_2 in these transitions are independent (we see that the order of μ_1 and μ_2 does not matter). In case we want to model $(\mu_1; \mu_2 + \mu_2; \mu_1)$ all transitions need to be dependent. $\square$

A useful property for non-atomic action refinement with LTSI, is the so called *diamond closure* ([DR95]). The term diamond refers to the form of the transition system in case of parallelism (and choice), the way the transitions t_1 to t_4 form a diamond in Figure 5.15. Without going into the formal details, the diamond property in short states the following:

- If there is a choice between two *independent* transitions starting in the same state, then these transitions are part of the same "diamond". We illustrate that with Figure 5.15: t_1 and t_2 start in the same state.

When these transitions are independent this means that they can be executed in arbitrary order. This means that there should be corresponding transitions, like t_3 and t_4, ending in the same state, as depicted in the figure.

- Two *independent* “sequential” transitions, i.e., transitions that share an intermediate state form a diamond. This means that if transition t_1 and t_3 are independent in Figure 5.15 on the preceding page, there should be transitions like t_2 and t_4 to form a diamond with.
- Two independent transitions that end in the same state are part of the same diamond. t_3 and t_4 in Figure 5.15 on the facing page are an example of this. When these two transitions are independent there should be corresponding transitions, like t_1 and t_2 to form a diamond, because the actions of t_3 and t_4 can be performed in arbitrary order.

Bednarczyk ([Bed88]) shows that ALTS models underlie a variety of non-interleaving models for concurrency. In particular they subsume so-called Mazurkiewicz trace languages and (prime) event structures. Together with the work of Rensink and Wehrheim ([RW01]) on dependencies this strengthens us in our believe that it should be possible to come up with an action refinement approach for LTSIs. The following example shows a *gedanken experiment* on action refinement with LTSI.

Example 5.8.3 In Figure 5.16 on the next page we show an LTSI (left side) that models $(a \parallel x); a; x$. In order to be able to refer to the states and transitions we added their identifiers. Table 5.1 gives the dependency relation for the system; we only record the dependency relation between transitions that have a common state. The table tells us that transitions t_1 and t_2 are independent (as are t_1 and t_3, and t_2 and t_4). Therefore we can conclude that the diamond that transitions t_1, t_2, t_3 and t_4 form represents parallel behavior and not choice behavior (in the case of choice, these transitions would be dependent and then the system would model

Transition	t_1	t_2	t_3	t_4	t_5	t_6
t_1 (q_0, a, q_1)		R	R			
t_2 (q_0, x, q_2)	R			R		
t_3 (q_1, x, q_3)	R			R	R_d	
t_4 (q_2, a, q_3)		R	R		R_d	
t_5 (q_3, a, q_4)			R_d	R_d		R_d
t_6 (q_4, x, q_5)					R_d	

Table 5.1: Dependency relation for abstract system

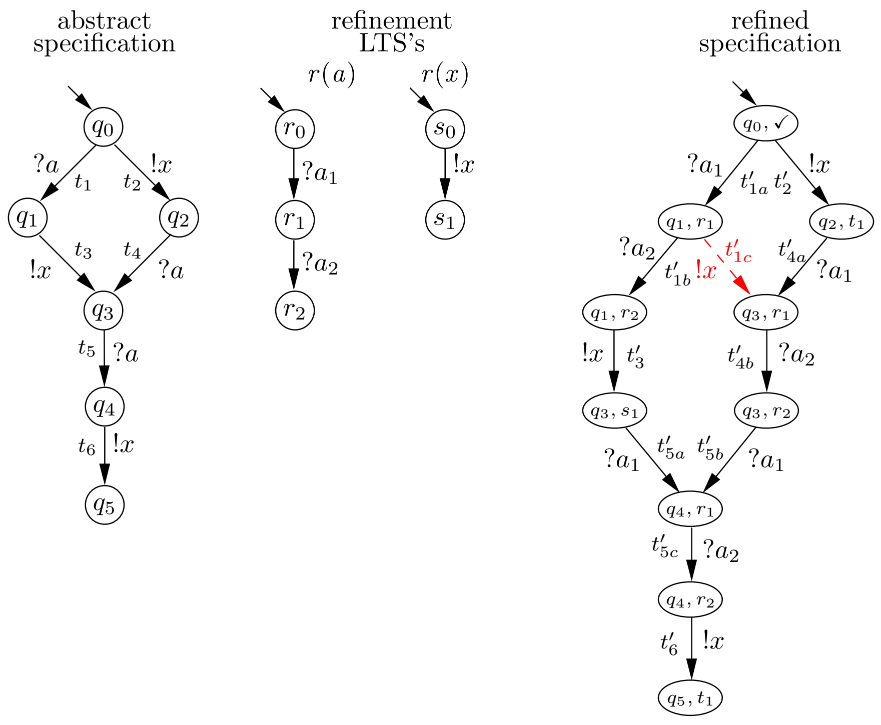


Figure 5.16: Non Atomic Refinement of LTSI

$(a; x + x; a); a; x)$. Transitions t_5 and t_6 are the result of sequential behavior and therefore dependent.

Now suppose we refine $a \to a_1; a_2$ and $x \to x$, like the refinement LTSs shown in the middle of Figure 5.16 (there's no concurrency in the refinement LTSs, so all transitions are dependent). When we refine the transition system with our atomic approach from Chapter 5 we obtain the transition system on the right-hand side in Figure 5.16 (without the dashed transition t'_{1c}); for easier reading we named the transitions in the refined system after the original transitions; for example transitions t'_{1a} and t'_{1b} have t_1 as their abstract original.

Because transitions t_1 and t_2 are independent we expect non-atomic action refinement to come up with (dashed) transition t'_{1c}. With the aid of the independence relation and the diamond property this should be possible. When we let the refinements inherit their dependency relation from the abstract transition, we know that t'_{1a} and t'_2 are independent and therefore they should form a diamond. With this piece of information we know that transition t'_{1c} should be added.

Notice that this system does not yet qualify as a proper LTSI. As $t'_{1b} R t'_{1c}$ and $t_{1b'} R t'_3$ we would expect t'_3 and t'_{4b} to end in the same state, thus forming a diamond with t'_{1b} and t'_{1c}. Because of the way we have defined atomic action refinement (q_3, s_1) and (q_3, r_2) are different states. In principle we

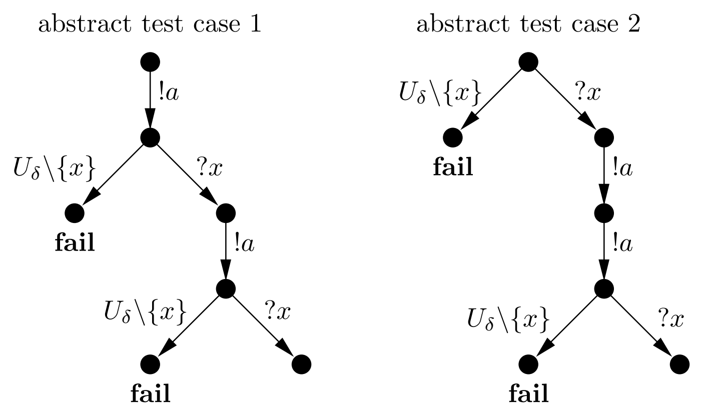

abstract test case 1
!a
$U_\delta\backslash\{x\}$
?x
fail
!a
$U_\delta\backslash\{x\}$
?x
fail
abstract test case 2
$U_\delta\backslash\{x\}$
?x
fail
!a
!a
$U_\delta\backslash\{x\}$
?x
fail

Figure 5.17: Abstract test-cases for non-atomic refinement example

can consider them to be the same state (it would result in a bisimilar system). For this example we could minimize the system by taking (q_3, s_1) and (q_3, r_2) together. Another approach might be to use a less strict form of the diamond closure. □

We strongly encourage further research on this topic. Once we have a proper LTSI we can generate test-cases with our existing test-case generation algorithm. However, non-atomic refinement of models is only half the story for action refinement in model-based testing. It allows us to refine models and (re)generate test-cases, but it leaves us still with the problem of test-case refinement.

5.8.2 Non-atomic test-case refinement

Like with atomic refinement we would like to have two ways to obtain a refined test suite for non-atomic refinement. One by first refining the abstract specification and generating refined test-cases from the refined specification, as we discussed above. The other approach is to refine an abstract test suite directly.

The issue with test-case refinement is that test-cases contain little information about the specification from which they were derived. This makes it difficult to assign verdicts for all possible observations and in some cases we can only assign the **inconclusive** verdict. For non-atomic refinement this lack of information only increases. We illustrate this with an example.

Example 5.8.4 We use the abstract and refined specifications of Figure 5.16 on the facing page; the abstract system modeling $(a \parallel x); a; x$ and the refined system modeling $(a_1; a_2 \parallel x); a_1; a_2; x$. In Figure 5.17 we show some abstract test-cases and in Figure 5.18 on the following page we show some refined test-cases. Suppose that we refine the abstract test-case with: $a \to a_1; a_2$

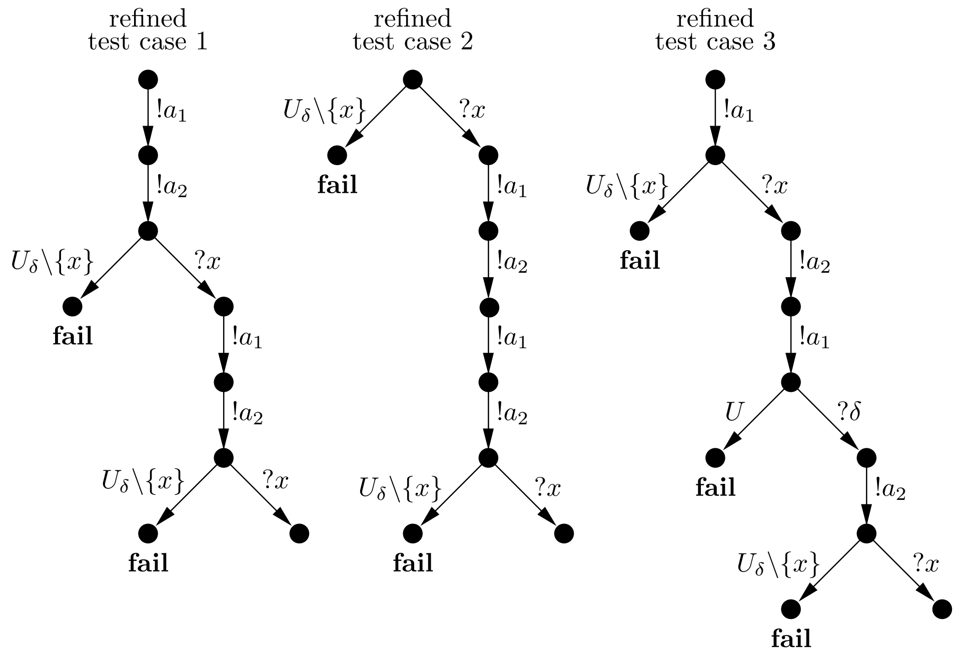


Figure 5.18: Refined test-cases for non-atomic refinement example

and $x \to x$, how do we obtain all possible non-atomically refined test-cases? For test-cases 1 and 2 this seems rather straightforward, as we can obtain them by atomic test-case refinement. For refined test-case 3, this is a different story. How do we know that the first occurrence of action x does not depend on a_2, whereas the second occurrence does? We do not have this kind of information in our test set, not even if it is complete. □

As we saw in the previous example we need more information to non-atomically refine test-cases. An interesting approach would be to add a dependency relation to test-cases, like in the case with specifications. We have to think of a new way to use the dependency relation; because of the linear nature of test-cases we cannot use the diamond closure.

5.8.3 Relaxed refinement

When we have knowledge about dependencies we can also apply this in our refinement relation. Relaxed refinement makes it possible to change the dependencies between refined actions. We illustrate relaxed refinement with the following example.

Example 5.8.5 In Figure 5.19 on the next page we show an LTS model for $x; a$ (left). In this case we refine $a \to a$ and $x \to x_1; x_2$. After refinement we obtain the refined model (middle). This model has the implicit assumption

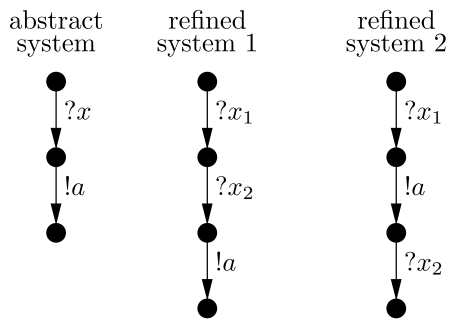


Figure 5.19: Relaxed refinement example

that because a depends on x, a also depends on x_1 and x_2. But what if we want a system where a is independent of x_2? On the right-hand side of Figure 5.19 we show the refinement of the abstract model with this kind of dependency. This more liberal use of dependencies is an example of relaxed refinement.

The reader may wonder if relaxed refinement is realistic, and whether it is a solution to action refinement problems in practice? Suppose that x is a remote procedure call, which we refine to a call part (x_1) and a response part (x_2). It depends on the functionality of the system whether a depends on x_2 or not; in other words whether a needs some information that x_2 provides. If x_1 writes some data to a database, or writes some logging data to a file, it can be perfectly fine that we do not want to wait for the response x_2 and want continue with a right away (for example for performance reasons). □

Like with non-atomic refinement we think that using a dependency relation together with (some form of) the diamond closure is an interesting possibility to research.

Example 5.8.6 In Figure 5.20 on the next page we show an abstract test-case (left-hand side) of the abstract transition system in Figure 5.19 (left-hand side). We want to refine the test-case such that it becomes a test-case for the refined system on the right-hand side of Figure 5.19. Refined test-case 1 is a straight forward (atomic) refinement, this should pose no problem. Test-case 2 is a refinement that takes the independence between a and x_2 into account. We somehow have to add this information to the test-case and/or refinement function and we need to have enough information to predict all allowed responses of the system. □

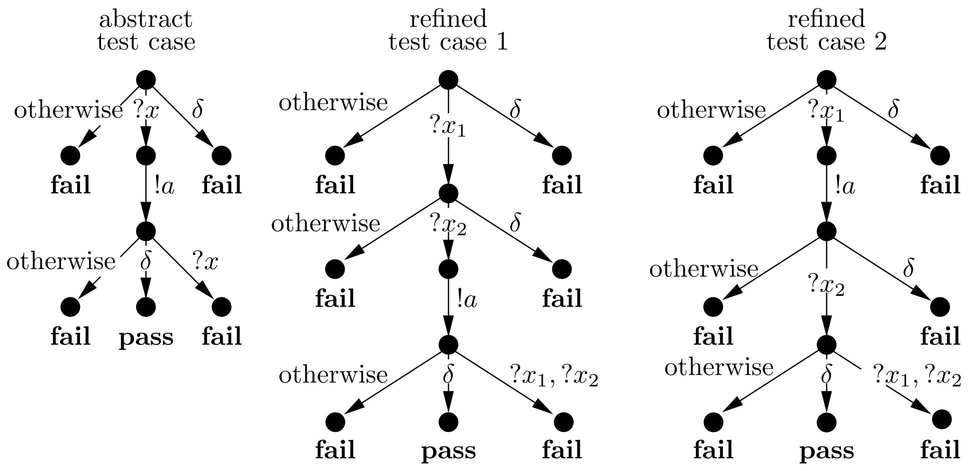


Figure 5.20: Relaxed test-case refinement example

Chapter 6

Concluding remarks

THE WORK IN THIS THESIS is centered around, what used to be, two open issues in model-based testing: compositional testing and action refinement in model-based testing. Both enable a more flexible use of models and test-cases in model-based testing. In this thesis we have explained our contributions to this research. Because the former chapters already contain conclusions specific to those chapters, we focus on the main contributions of this thesis and some ideas for further research.

We started with an overview of the research in model-based testing in Chapter 2. We defined and explained the testing concepts that we use throughout the thesis and we treated the **ioco**-theory, together with some theories that influenced it.

We find that an important idea, though somewhat hidden, behind **ioco** is: test what is specified and leave the non-specified parts to the implementation freedom of the developer. This means that if it is important to test the non-specified parts, someone should specify them first. While this idea and its implementation in **ioco** works for systems without parallel behavior, we showed in Chapter 3 that it does not work as expected in compositional testing: **ioco** is not suitable for compositional testing, because the **ioco** property is not preserved when testing communicating components. The main reason for this is that **ioco** allows underspecification of input actions. We showed that when specifications are modeled by IOTSs, **ioco** is suitable for compositional testing. In order to be able to test with models that are underspecified for input actions, we proposed three approaches. Our first approach was to make an LTS model input complete by with demonic completion, such that it captures our notion of underspecification. Our second investigation was to change the **ioco** implementation relation to **uioco**. Our final solution was to change the semantics of the parallel composition operator. With this result it has become possible to test systems that consist of several parallel components, with the **ioco** theory. Furthermore we have shed some light on the, in our opinion, hidden meaning and semantics of underspecification in the specification and input-enabledness in the IUT.

A constraint that we ran into in Chapter 3 is that the **ioco** theory assumes specifications to be strongly convergent. It would be very helpful if this constraint is removed in the future. We do not see any technical reasons prohibiting this, see for example the way τ-loops are treated in [JJ05]. It may involve quite some work, as some of the **ioco** proofs need to be checked and/or redone.

The next chapter, Chapter 4, introduced the area of action refinement in model-based testing. We presented our own action refinement approach in Chapter 5. We tried to put as little constraints as possible on our action refinement function. The main constraint is our limitation to atomic refinement. In the chapter we showed how to refine traces, transition systems and test-cases. We showed that direct refinement of a test suite generated from an abstract model is equivalent to first refining the model and regenerating the test suite. We concluded the chapter with suggestions for further research in the area of non-atomic and relaxed test-case refinement.

Looking back, it was rather unexpected how hard it was to apply action refinement to model-based testing. Especially the fact that test-cases do not have enough information to facilitate (non) atomic action refinement. After reading our work this may come as a surprise, as it seems rather obvious now.[1]. This is especially surprising since many of the tests used in practice are abstract tests, that require refinement of the actions before they can be used in test-execution. Contrary to our work, these refinements are generally done by humans, but after reading our work, it makes one wonder what is actually tested and how many errors remain hidden.

While doing our research on action refinement we looked into several test-case definitions for inspiration for a richer test-case formalism. Based on this exercise, we feel that model-based testing research would benefit from further work on test-cases and their definition. The main focus in model-based testing research seems to be on test-case derivation from a formal model. Other aspects, like for example the relation to the original model (important for our work), and the recording of what happened during test-case execution seem neglected. For example with Tretmans' definition of a test-case [Tre08] it is not possible to define what trace was actually executed against the IUT. This is not an exception, see for example the test-case definitions of [JJ05]) or [LY96] and [PY00] in the FSM research. Another aspect is that we can identify at least three abstraction levels in a test-case: the abstract test-case in terms of the model, the instantiated test-case (in which actions are refined and abstract data types are made concrete) and the concrete test-case (with actions in terms of IUT). This does not fit with the current test-case definitions.

[1]Quoting soccer player Johan Cruijff: "Je gaat het pas zien als je het doorhebt." (You only see it, when you're aware of it.)

Appendix A

Proofs of Chapter 3: Compositional testing with ioco

A.1 Proofs of Section 3.3.1: Parallel composition

In the proofs we use the notation L_p^δ to denote $L_p \cup \{\delta\}$ for a system p.

Proposition A.1.1 Let $p \in \mathbf{IOTS}(I_p, U_p),\ q \in \mathbf{IOTS}(I_q, U_q), r \in Q_{p\|q}$ with $I_p \cap I_q = U_p \cap U_q = \emptyset$.

1. $\forall a \in L_p \setminus L_q : p\|q \xrightarrow{a} r \Leftrightarrow \exists p' : p \xrightarrow{a} p' \wedge r = p'\|q$
2. $\forall a \in L_q \setminus L_p : p\|q \xrightarrow{a} r \Leftrightarrow \exists q' : q \xrightarrow{a} q' \wedge r = p\|q'$
3. $p\|q \xrightarrow{\tau} r \Leftrightarrow (\exists p' : p \xrightarrow{\tau} p' \wedge r = p'\|q) \vee (\exists q' : q \xrightarrow{\tau} q' \wedge r = p\|q')$
4. $\forall a \in (L_p \cap L_q) \cup \{\delta\} : p\|q \xrightarrow{a} r \Leftrightarrow \exists p', q' : p \xrightarrow{a} p' \wedge q \xrightarrow{a} q' \wedge r = p'\|q'$

$\square$

Proof

1. Assume $a \in L_p \setminus L_q$

   only if:
   $$\begin{array}{ll} & p\|q \xrightarrow{a} r \\ \Rightarrow & (*\ \text{definition}\ \|\ *) \\ & \exists p' : p \xrightarrow{a} p' \wedge r = p'\|q \end{array}$$

   if:
   $$\begin{array}{ll} & \exists p' : p \xrightarrow{a} p' \wedge r = p'\|q \\ \Rightarrow & (*\ \text{definition}\ \|\ *) \\ & p\|q \xrightarrow{a} r \end{array}$$

2. Analogous to 1
3. Analogous to 1
4. The case $a \in L_p \cap L_q$ is analogous to 1. Here the proof for $a = \delta$ is given.

only if: First we show $r = p\|q$, then the remaining part of the proof is given.

$$\begin{aligned} & p\|q \xrightarrow{\delta} r \\ \Rightarrow\ & (*\ \text{Definition } \delta\ \ *) \\ & p\|q \xrightarrow{\delta} r \wedge r = p\|q \end{aligned} \tag{A.1}$$

$$\begin{aligned}
& p\|q \xrightarrow{\delta} r \\
\Rightarrow\ & (*\ \text{Definition } \delta\ \ *) \\
& \forall \mu \in U_p \cup U_q \cup \{\tau\} : p\|q \not\xrightarrow{\mu} \\
\Rightarrow\ & (*\ \text{Definition } \|\ \ *) \\
& p \not\xrightarrow{\tau} \ \wedge q \not\xrightarrow{\tau} \\
& \wedge \forall \mu \in (U_p \backslash L_q) : p \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in (U_q \backslash L_p) : q \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in L_p \cap L_q : p \not\xrightarrow{\mu} \ \vee q \not\xrightarrow{\mu} \\
\Rightarrow\ & (*\ p, q \in \textbf{IOTS}\ \ \ *) \\
& p \not\xrightarrow{\tau} \ \wedge q \not\xrightarrow{\tau} \\
& \wedge \forall \mu \in (U_p \backslash L_q) : p \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in (U_q \backslash L_p) : q \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in L_p \cap L_q : p \not\xrightarrow{\mu} \ \vee q \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in I_p : p \xRightarrow{\mu} \wedge \forall \mu \in I_q : q \xRightarrow{\mu} \\
\Rightarrow\ & (*\ p, q \not\xrightarrow{\tau}\ \ \ \ *) \\
& p \not\xrightarrow{\tau} \ \wedge q \not\xrightarrow{\tau} \\
& \wedge \forall \mu \in (U_p \backslash L_q) : p \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in (U_q \backslash L_p) : q \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in L_p \cap L_q : p \not\xrightarrow{\mu} \ \vee q \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in I_p : p \xrightarrow{\mu} \wedge \forall \mu \in I_q : q \xrightarrow{\mu} \\
\Rightarrow\ & (*\ \text{Definition } \textbf{IOTS}\text{: } I \cap U = \emptyset\ \ *) \\
& p \not\xrightarrow{\tau} \ \wedge q \not\xrightarrow{\tau} \\
& \wedge \forall \mu \in (U_p \backslash L_q) : p \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in (U_q \backslash L_p) : q \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in U_p \cap L_q : p \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in L_p \cap U_q : q \not\xrightarrow{\mu} \\
\Rightarrow\ & (*\ \text{Set operations}\ \ *) \\
& p \not\xrightarrow{\tau} \ \wedge q \not\xrightarrow{\tau} \\
& \wedge \forall \mu \in U_p : p \not\xrightarrow{\mu} \\
& \wedge \forall \mu \in U_q : q \not\xrightarrow{\mu} \\
\Rightarrow\ & (*\ \text{definition } \delta \text{ and } r = p\|q \text{ (see A.1)}\ \ *) \\
& p \xrightarrow{\delta} p \wedge q \xrightarrow{\delta} q \wedge r = p\|q
\end{aligned}$$

if: Let $r = p\|q$

$$\begin{array}{ll}
& p \xrightarrow{\delta} p \wedge q \xrightarrow{\delta} q \\
\Rightarrow & (*\ \text{definition } \delta\ *) \\
& \forall \mu \in U_p \cup \{\tau\} : p \overset{\mu}{\not\longrightarrow} \wedge \forall \mu \in U_q \cup \{\tau\} : q \overset{\mu}{\not\longrightarrow} \\
\Rightarrow & (*\ \text{definition } \|\ *) \\
& \forall \mu \in U_p \cup U_q \cup \{\tau\} : p\|q \overset{\mu}{\not\longrightarrow} \\
\Rightarrow & (*\ \text{definition } \delta\ *) \\
& p\|q \xrightarrow{\delta} p\|q \\
\Rightarrow & (*\ \text{Given: } r = p\|q\ *) \\
& p\|q \xrightarrow{\delta} r
\end{array}$$

□

Proposition A.1.2 Let $p \in \mathbf{IOTS}(I_p, U_p)$, $q \in \mathbf{IOTS}(I_q, U_q)$, $r \in Q_{p\|q}$, $\sigma \in L_\delta^*$, with $L = I_p \cup I_q \cup U_p \cup U_q$ and $I_p \cap I_q = U_p \cap U_q = \emptyset$.

$$p\|q \xRightarrow{\sigma} r \Leftrightarrow \exists p', q' : p \xRightarrow{\sigma \lceil L_p^\delta} p' \wedge q \xRightarrow{\sigma \lceil L_q^\delta} q' \wedge r = p'\|q'$$

□

Proof

Only if: Proof by induction on the structure of σ.

Basic step: $\sigma = \epsilon$.

$$\begin{array}{ll}
& p\|q \xRightarrow{\epsilon} r \\
\Rightarrow & (*\ \text{Proposition A.1.1.3 and definition of } \xRightarrow{\epsilon}\ *) \\
& \exists p', q' : p \xRightarrow{\epsilon} p' \wedge q \xRightarrow{\epsilon} q' \wedge r = p'\|q' \\
\Rightarrow & (*\ \text{Definition 2.3.5 of projection}\ *) \\
& \exists p', q' : p \xRightarrow{\epsilon \lceil L_p^\delta} p' \wedge q \xRightarrow{\epsilon \lceil L_q^\delta} q' \wedge r = p'\|q'
\end{array}$$

Induction step: We make the assumption that the proposition holds for σ' in $\sigma = a \cdot \sigma'$, with $a \in L_\delta$. We identify three cases:

1. Assume $a \in L_p^\delta \cap L_q^\delta$ $(= (L_p \cap L_q) \cup \{\delta\})$.

$$
\begin{array}{ll}
 & p \| q \xRightarrow{a \cdot \sigma'} r \\
\Rightarrow & (*\ \text{Definition } \xRightarrow{a \cdot \sigma'}\ *) \\
 & \exists r_1, r_2 : p\|q \xRightarrow{\epsilon} r_1 \xrightarrow{a} r_2 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Basic step}\ *) \\
 & \exists p_1, q_1, r_2 : p \xRightarrow{\epsilon} p_1 \wedge q \xRightarrow{\epsilon} q_1 \wedge p_1\|q_1 \xrightarrow{a} r_2 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Proposition A.1.1.4}\ *) \\
 & \exists p_1, p_2, q_1, q_2 : p \xRightarrow{\epsilon} p_1 \wedge q \xRightarrow{\epsilon} q_1 \wedge p_1 \xrightarrow{a} p_2 \wedge q_1 \xrightarrow{a} q_2 \\
 & \wedge\, p_2\|q_2 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Definition } \xRightarrow{a}\ *) \\
 & \exists p_2, q_2 : p \xRightarrow{a} p_2 \wedge q \xRightarrow{a} q_2 \wedge p_2\|q_2 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists p', p_2, q', q_2 : p \xRightarrow{a} p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{a} q_2 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
 & \wedge\, r = p'\|q' \\
\Rightarrow & (*\ \text{Definition 2.3.5 of projection}\ *) \\
 & \exists p', q' : p \xRightarrow{(a \cdot \sigma') \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{(a \cdot \sigma') \upharpoonright L_q^\delta} q' \wedge r = p'\|q' \\
\Rightarrow & (*\ \sigma = a \cdot \sigma'\ *) \\
 & \exists p', q' : p \xRightarrow{\sigma \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{\sigma \upharpoonright L_q^\delta} q' \wedge r = p'\|q'
\end{array}
$$

2. Assume $a \in L_p^\delta \setminus L_q^\delta$ $(= L_p \backslash L_q)$.

$$
\begin{array}{ll}
 & p \| q \xRightarrow{a \cdot \sigma'} r \\
\Rightarrow & (*\ \text{Definition } \xRightarrow{\sigma}\ *) \\
 & \exists r_1, r_2 : p\|q \xRightarrow{\epsilon} r_1 \xrightarrow{a} r_2 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Basic step}\ *) \\
 & \exists p_1, q_1, r_2 : p \xRightarrow{\epsilon} p_1 \wedge q \xRightarrow{\epsilon} q_1 \wedge p_1\|q_1 \xrightarrow{a} r_2 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Proposition A.1.1.1}\ *) \\
 & \exists p_1, p_2, q_1 : p \xRightarrow{\epsilon} p_1 \wedge q \xRightarrow{\epsilon} q_1 \wedge p_1 \xrightarrow{a} p_2 \wedge p_2\|q_1 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Definition of } \xRightarrow{a}\ *) \\
 & \exists p_2, q_1 : p \xRightarrow{a} p_2 \wedge q \xRightarrow{\epsilon} q_1 \wedge p_2\|q_1 \xRightarrow{\sigma'} r \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists p_2, p', q_1, q' : p \xRightarrow{a} p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{\epsilon} q_1 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
 & \wedge\, r = p'\|q' \\
\Rightarrow & (*\ \text{Definition 2.3.5 of projection}\ *) \\
 & \exists p', q' : p \xRightarrow{(a \cdot \sigma') \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{(a \cdot \sigma') \upharpoonright L_q^\delta} q' \wedge r = p'\|q' \\
\Rightarrow & (*\ \sigma = a \cdot \sigma'\ *) \\
 & \exists p', q' : p \xRightarrow{\sigma \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{\sigma \upharpoonright L_q^\delta} q' \wedge r = p'\|q'
\end{array}
$$

3. Assume $a \in L_q^\delta \setminus L_p^\delta$. This is symmetric with the previous case.

if: By induction on the structure of σ.

Basic step: $\sigma = \epsilon$.

$$
\begin{array}{ll}
 & \exists p', q' : p \xRightarrow{\epsilon \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{\epsilon \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Definition 2.3.5 of projection}\ *) \\
 & \exists p', q' : p \xRightarrow{\epsilon} p' \wedge q \xRightarrow{\epsilon} q' \\
\Rightarrow & (*\ \text{Proposition A.1.1.3 and definition of } \xRightarrow{\epsilon}\ *) \\
 & \exists p', q' : p \| q \xRightarrow{\epsilon} p' \| q' \\
\Rightarrow & (*\ \sigma = \epsilon\ *) \\
 & \exists p', q' : p \| q \xRightarrow{\sigma} p' \| q' \\
\Rightarrow & (*\ \text{Given } r = p' \| q'\ *) \\
 & p \| q \xRightarrow{\sigma} r
\end{array}
$$

Induction step: We assume the proposition holds for σ' in $\sigma = a \cdot \sigma'$, $a \in L_\delta$. We identify three cases:

1. Assume $a \in L_p^\delta \cap L_q^\delta$ $(= (L_p \cap L_q) \cup \{\delta\})$:

$$
\begin{array}{ll}
 & \exists p', q' : p \xRightarrow{(a \cdot \sigma') \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{(a \cdot \sigma') \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Definition 2.3.5 of projection}\ *) \\
 & \exists p_2, p', q_2, q' : p \xRightarrow{a} p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{a} q_2 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Definition of } \Longrightarrow\ *) \\
 & \exists p_1, p_2, p', q_1, q_2, q' : \\
 & p \xRightarrow{\epsilon} p_1 \xrightarrow{a} p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q \xRightarrow{\epsilon} q_1 \xrightarrow{a} q_2 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Basic step}\ *) \\
 & \exists p_1, p_2, p', q_1, q_2, q' : \\
 & p \| q \xRightarrow{\epsilon} p_1 \| q_1 \wedge p_1 \xrightarrow{a} p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q_1 \xrightarrow{a} q_2 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Proposition A.1.1.4}\ *) \\
 & \exists p_1, p_2, p', q_1, q_2, q' : \\
 & p \| q \xRightarrow{\epsilon} p_1 \| q_1 \xrightarrow{a} p_2 \| q_2 \wedge p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q_2 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Definition of } \Longrightarrow\ *) \\
 & \exists p_2, p', q_2, q' : p \| q \xRightarrow{a} p_2 \| q_2 \wedge p_2 \xRightarrow{\sigma' \upharpoonright L_p^\delta} p' \wedge q_2 \xRightarrow{\sigma' \upharpoonright L_q^\delta} q' \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists p_2, p', q_2, q' : p \| q \xRightarrow{a} p_2 \| q_2 \wedge p_2 \| q_2 \xRightarrow{\sigma'} p' \| q' \\
\Rightarrow & (*\ \text{Definition of } \Longrightarrow\ *) \\
 & \exists p', q' : p \| q \xRightarrow{a \cdot \sigma'} p' \| q' \\
\Rightarrow & (*\ \sigma = a \cdot \sigma'\ *) \\
 & \exists p', q' : p \| q \xRightarrow{\sigma} p' \| q' \\
\Rightarrow & (*\ \text{Given } r = p' \| q'\ *) \\
 & p \| q \xRightarrow{\sigma} r
\end{array}
$$

2. Assume $a \in L_p^\delta \setminus L_q^\delta$ $(= L_p \backslash L_q)$:
$\exists p', q' : p \xRightarrow{(a \cdot \sigma') \lceil L_p^\delta} p' \wedge q \xRightarrow{(a \cdot \sigma') \lceil L_q^\delta} q'$
$\Rightarrow$ (∗ Definition 2.3.5 of projection ∗)
$\exists p_2, p', q' : p \xRightarrow{a} p_2 \xRightarrow{\sigma' \lceil L_p^\delta} p' \wedge q \xRightarrow{\sigma' \lceil L_q^\delta} q'$
$\Rightarrow$ (∗ Definition of $\Longrightarrow$ ∗)
$\exists p_1, p_2, p', q' : p \xRightarrow{\epsilon} p_1 \xrightarrow{a} p_2 \xRightarrow{\sigma' \lceil L_p^\delta} p' \wedge q \xRightarrow{\sigma' \lceil L_q^\delta} q'$
$\Rightarrow$ (∗ Basic step ∗)
$\exists p_1, p_2, p', q' : p \| q \xRightarrow{\epsilon} p_1 \| q \wedge p_1 \xrightarrow{a} p_2 \xRightarrow{\sigma' \lceil L_p^\delta} p'$
$\wedge\, q \xRightarrow{\sigma' \lceil L_q^\delta} q'$
$\Rightarrow$ (∗ Proposition A.1.1.1 ∗)
$\exists p_1, p_2, p', q' : p \| q \xRightarrow{\epsilon} p_1 \| q \xrightarrow{a} p_2 \| q \wedge p_2 \xRightarrow{\sigma' \lceil L_p^\delta} p'$
$\wedge\, q \xRightarrow{\sigma' \lceil L_q^\delta} q'$
$\Rightarrow$ (∗ Definition of $\Longrightarrow$ ∗)
$\exists p_2, p', q' : p \| q \xRightarrow{a} p_2 \| q \wedge p_2 \xRightarrow{\sigma' \lceil L_p^\delta} p' \wedge q \xRightarrow{\sigma' \lceil L_q^\delta} q'$
$\Rightarrow$ (∗ Induction ∗)
$\exists p_2, p', q' : p \| q \xRightarrow{a} p_2 \| q \wedge p_2 \| q \xRightarrow{\sigma'} p' \| q'$
$\Rightarrow$ (∗ Definition of $\Longrightarrow$ ∗)
$\exists p', q' : p \| q \xRightarrow{a \cdot \sigma'} p' \| q'$
$\Rightarrow$ (∗ $\sigma = a \cdot \sigma'$ ∗)
$\exists p', q' : p \| q \xRightarrow{\sigma} p' \| q'$
$\Rightarrow$ (∗ Given $r = p' \| q'$ ∗)
$p \| q \xRightarrow{\sigma} r$

3. Assume $a \in L_q^\delta \setminus L_p^\delta$. This is symmetric with the previous case.

□

Lemma A.1.3 Let $i, s \in \mathbf{IOTS}(I, U)$, then:

$$i \ \mathbf{ioco}\ s \quad \Leftrightarrow \quad \mathit{Straces}(i) \ \subseteq \ \mathit{Straces}(s)$$

□

Proof

only if: Proof by induction on the structure of σ, let $\sigma \in \mathit{Straces}(i)$.

Basic Step: $\sigma = \epsilon$.
$\epsilon \in \mathit{Straces}(s)$ trivially holds.

Induction step : We identify two cases:

1. $\sigma = \rho \cdot a$ with $a \in I$:
By induction $\rho \in \mathit{Straces}(s)$, so $\exists s' : s \xRightarrow{\rho} s'$, and since $s \in \mathbf{IOTS}(I, U)$, $s' \xRightarrow{a}$ always holds. Hence, $\rho \cdot a \in \mathit{Straces}(s)$.

2. $\sigma = \rho{\cdot}x$ with $x \in U \cup \{\delta\}$:
   If $i \xRightarrow{\rho\cdot x}$ then by definition of **ioco**: $x \in out(i \textbf{ after } \rho)$, and since i **ioco** s and by induction $\rho \in Straces(s)$ we can conclude that $x \in out(s \textbf{ after } \rho)$. Hence, $s \xRightarrow{\rho\cdot x}$, and $\rho \cdot x \in Straces(s)$.

if: Let $\sigma \in Straces(s)$ and $x \in out(i \textbf{ after } \sigma)$, then $i \xRightarrow{\sigma\cdot x}$, which implies $\sigma{\cdot}x \in Straces(i)$, hence $\sigma{\cdot}x \in Straces(s)$ and $s \xRightarrow{\sigma\cdot x}$, from which it follows that $x \in out(s \textbf{ after } \sigma)$.

□

Theorem 3.3.3 Let $s_1, i_1 \in \mathbf{IOTS}(I_1, U_1), s_2, i_2 \in \mathbf{IOTS}(I_2, U_2)$ with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$.

$$i_1 \textbf{ ioco } s_1 \wedge i_2 \textbf{ ioco } s_2 \Rightarrow i_1 \| i_2 \textbf{ ioco } s_1 \| s_2$$

□

Proof To be proved according to Lemma A.1.3:

$$\begin{aligned} &Straces(i_1) \subseteq Straces(s_1) \wedge Straces(i_2) \subseteq Straces(s_2) \\ &\quad \Rightarrow Straces(i_1 \| i_2) \subseteq Straces(s_1 \| s_2) \end{aligned}$$

$$\begin{array}{ll} & \sigma \in Straces(i_1 \| i_2) \\ \Rightarrow & (*\text{ Definition of } Straces \;*) \\ & i_1 \| i_2 \xRightarrow{\sigma} \\ \Rightarrow & (*\text{ Proposition A.1.2 } *) \\ & i_1 \xRightarrow{\sigma \upharpoonright L^{\delta}_{i_1}} \wedge i_2 \xRightarrow{\sigma \upharpoonright L^{\delta}_{i_2}} \\ \Rightarrow & (*\text{ Premise: } L_{i_1} = L_{s_1}, L_{i_2} = L_{s_2} \;*) \\ & s_1 \xRightarrow{\sigma \upharpoonright L^{\delta}_{s_1}} \wedge s_2 \xRightarrow{\sigma \upharpoonright L^{\delta}_{s_2}} \\ \Rightarrow & (*\text{ Proposition A.1.2 } *) \\ & s_1 \| s_2 \xRightarrow{\sigma} \\ \Rightarrow & (*\text{ Definition of } Straces \;*) \\ & \sigma \in Straces(s_1 \| s_2) \end{array}$$

□

A.2 Proofs of Section 3.3.2: Hiding

Definition A.2.1 The hiding of $\sigma \in L_\delta^*$ with $A \subseteq L$, denoted as $\sigma \backslash A$, is defined as follows:

$$\sigma\backslash A \;=_{\text{def}} \begin{cases} \epsilon & \sigma = \epsilon \\ \sigma'\backslash A & \sigma = a\cdot\sigma' \text{ with } a \in A \\ a\cdot(\sigma'\backslash A) & \sigma = a\cdot\sigma' \text{ with } a \notin A \\ \delta\cdot(\sigma'\backslash A) & \sigma = \delta\cdot\sigma' \end{cases}$$

Proposition A.2.2 Let $p \in \mathbf{LTS}(I, U)$ where U is partitioned in sets U_1 and U_2.

1. $p \xrightarrow{a} p'$ with $a \in I \cup U_1 \;\Rightarrow\; \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{a} \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
2. $p \xrightarrow{a} p'$ with $a \in U_2 \;\Rightarrow\; \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau} \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
3. $p \xrightarrow{\tau} p' \;\Rightarrow\; \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau} \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
4. $p \xrightarrow{\delta} p' \;\Rightarrow\; \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\delta} \mathbf{hide}\, U_2\, \mathbf{in}\, p$
5. $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{a} q$ with $a \in I \cup U_1 \;\Rightarrow\; \exists p' :\; p \xrightarrow{a} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
6. $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau} q \;\Rightarrow\; (\; \exists p' :\; p \xrightarrow{\tau} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \;)$
   $\vee\;\; (\; \exists p', a \in U_2 :\; p \xrightarrow{a} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \;)$
7. $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\delta} q \;\Rightarrow\; p \xrightarrow{\delta} p \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p$

□

Proof

1. $p \xrightarrow{a} p' \wedge a \in I \cup U_1$
   $\Rightarrow$ (∗ Definition of hide ∗)
   $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{a} \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
2. $p \xrightarrow{a} p' \wedge a \in U_2$
   $\Rightarrow$ (∗ Definition of hide ∗)
   $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau} \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
3. $p \xrightarrow{\tau} p'$
   $\Rightarrow$ (∗ Definition of hide ∗)
   $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau} \mathbf{hide}\, U_2\, \mathbf{in}\, p'$
4. $p \xrightarrow{\delta} p'$
   $\Rightarrow$ (∗ Definition of quiescence ∗)
   $(\; \forall \mu \in U \cup \{\tau\} :\; p \overset{\mu}{\not\longrightarrow} \;)$
   $\Rightarrow$ (∗ Definition of hide ∗)
   $(\; \forall \mu \in U_1 \cup \{\tau\} :\; \mathbf{hide}\, U_2\, \mathbf{in}\, p \overset{\mu}{\not\longrightarrow} \;)$
   $\Rightarrow$ (∗ Definition of quiescence ∗)
   $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\delta} \mathbf{hide}\, U_2\, \mathbf{in}\, p$
5. $\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{a} q$
   $\Rightarrow$ (∗ Definition of hide ∗)
   $\exists p' :\; p \xrightarrow{a} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p'$

6. $\quad \textbf{hide}\, U_2\, \textbf{in}\, p \xrightarrow{\tau} q$
$\Rightarrow$ $(*$ Definition of hide $*)$
$(\ \exists p' :\ p \xrightarrow{\tau} p' \wedge q = \textbf{hide}\, U_2\, \textbf{in}\, p'\ )\ \vee$
$(\ \exists p', \exists a \in U_2 :\ p \xrightarrow{a} p' \wedge q = \textbf{hide}\, U_2\, \textbf{in}\, p'\ )$

7. $\quad \textbf{hide}\, U_2\, \textbf{in}\, p \xrightarrow{\delta} q$
$\Rightarrow$ $(*$ Definition of quiescence $*)$
$\forall \mu \in U_1 \cup \{\tau\} :\ \textbf{hide}\, U_2\, \textbf{in}\, p \not\xrightarrow{\mu}\ \wedge\ q = \textbf{hide}\, U_2\, \textbf{in}\, p$
$\Rightarrow$ $(*$ Definition of hide $*)$
$(\ \forall \mu \in U_1 :\ p \not\xrightarrow{\mu}\ )\ \wedge\ (\ \forall \mu \in U_2 :\ p \not\xrightarrow{\mu}\ )\ \wedge$
$p \not\xrightarrow{\tau}\ \wedge\ q = \textbf{hide}\, U_2\, \textbf{in}\, p$
$\Rightarrow$ $(*$ Logical axioms $*)$
$\forall \mu \in U \cup \{\tau\} :\ p \not\xrightarrow{\mu}\ \wedge\ q = \textbf{hide}\, U_2\, \textbf{in}\, p$
$\Rightarrow$ $(*$ Definition of quiescence $*)$
$p \xrightarrow{\delta} p \wedge q = \textbf{hide}\, U_2\, \textbf{in}\, p$

□

Proposition A.2.3 Let $p \in \mathbf{LTS}(I, U)$ where U is partitioned into U_1 and U_2; let $\sigma \in L_\delta^*$ be arbitrary.
1. $p \overset{\sigma}{\Longrightarrow} p' \ \Rightarrow\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\sigma \backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$
2. $\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\sigma}{\Longrightarrow} q \ \Rightarrow\ \exists p',\ \exists \sigma' \in L_\delta^* :\ p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \textbf{hide}\, U_2\, \textbf{in}\, p' \wedge \sigma = \sigma' \backslash U_2$

□

Proof
1. By induction on the structure of σ:
$\sigma = \epsilon$:
Using $\epsilon \backslash U_2 = \epsilon$, the proposition reduces to:

$$p \overset{\epsilon}{\Longrightarrow} p' \ \Rightarrow\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p' \tag{A.2}$$

which, using the definition of $\overset{\epsilon}{\Longrightarrow}$, is rewritten to:

$$p \xrightarrow{\tau^n} p' \ \Rightarrow\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p' \tag{A.3}$$

which is proved by induction on n:
$n = 0$:
$p \xrightarrow{\tau^n} p'$
$\Rightarrow$ $(*$ $n = 0$ and from definition of $\overset{\epsilon}{\Longrightarrow}$: $p \xrightarrow{\tau^0} p' \Leftrightarrow p = p'$ $*)$
$p = p'$
$\Rightarrow$ $(*$ definition of $\overset{\epsilon}{\Longrightarrow}$ $*)$
$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$n = n' + 1$:

$p \xrightarrow{\tau^n} p'$

$\Rightarrow$ (* $n = n' + 1$ and definition of $\overset{\epsilon}{\Longrightarrow}$ *)

$\exists p_1 :\ p \xrightarrow{\tau} p_1 \wedge p_1 \xrightarrow{\tau^{n'}} p'$

$\Rightarrow$ (* Proposition A.2.2.3 *)

$\textbf{hide}\, U_2\, \textbf{in}\, p \xrightarrow{\tau} \textbf{hide}\, U_2\, \textbf{in}\, p_1 \wedge p_1 \xrightarrow{\tau^{n'}} p'$

$\Rightarrow$ (* induction on equation (A.3) *)

$\textbf{hide}\, U_2\, \textbf{in}\, p \xrightarrow{\tau} \textbf{hide}\, U_2\, \textbf{in}\, p_1 \wedge \textbf{hide}\, U_2\, \textbf{in}\, p_1 \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (* definition of $\overset{\epsilon}{\Longrightarrow}$ *)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\sigma = a{\cdot}\rho$, with $a \in I \cup U_1$, $\rho \in L_\delta^*$:

$p \overset{\sigma}{\Longrightarrow} p'$

$\Rightarrow$ (* $\sigma = a{\cdot}\rho$ *)

$p \overset{a\cdot\rho}{\Longrightarrow} p'$

$\Rightarrow$ (* definition of $\overset{a\cdot\rho}{\Longrightarrow}$ *)

$\exists p_1, p_2 :\ p \overset{\epsilon}{\Longrightarrow} p_1 \wedge p_1 \xrightarrow{a} p_2 \wedge p_2 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (* equation (A.2) *)

$\exists p_1, p_2 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1 \wedge p_1 \xrightarrow{a} p_2 \wedge p_2 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (* Proposition A.2.2.1 *)

$\exists p_1, p_2 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \xrightarrow{a} \textbf{hide}\, U_2\, \textbf{in}\, p_2 \wedge p_2 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (* induction *)

$\exists p_1, p_2 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \xrightarrow{a} \textbf{hide}\, U_2\, \textbf{in}\, p_2 \wedge$

$\textbf{hide}\, U_2\, \textbf{in}\, p_2 \overset{\rho\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (* definition of $\overset{a\cdot(\rho\backslash U_2)}{\Longrightarrow}$ *)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{a\cdot(\rho\backslash U_2)}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (* Definition A.2.1 *)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{(a\cdot\rho)\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (* $\sigma = a{\cdot}\rho$ *)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\sigma\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\sigma = a\cdot\rho$, with $a \in U_2, \rho \in L_\delta^*$:

$p \overset{\sigma}{\Longrightarrow} p'$

$\Rightarrow$ (∗ $\sigma = a\cdot\rho$ ∗)

$p \overset{a\cdot\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ definition $\overset{a\cdot\rho}{\Longrightarrow}$ ∗)

$\exists p_1, p_2 :\ p \overset{\epsilon}{\Longrightarrow} p_1 \wedge p_1 \xrightarrow{a} p_2 \wedge p_2 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ equation (A.2) ∗)

$\exists p_1, p_2 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1 \wedge p_1 \xrightarrow{a} p_2 \wedge p_2 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ Proposition A.2.2.2 ∗)

$\exists p_1, p_2 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \xrightarrow{\tau} \textbf{hide}\, U_2\, \textbf{in}\, p_2 \wedge p_2 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ induction ∗)

$\exists p_1, p_2 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \xrightarrow{\tau} \textbf{hide}\, U_2\, \textbf{in}\, p_2$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_2 \overset{\rho\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (∗ definition of $\overset{\rho\backslash U_2}{\Longrightarrow}$ ∗)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\rho\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (∗ Definition A.2.1 ∗)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{(a\cdot\rho)\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (∗ $\sigma = a\cdot\rho$ ∗)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\sigma\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\sigma = \delta\cdot\rho$, with $\rho \in L_\delta^*$:

$p \overset{\sigma}{\Longrightarrow} p'$

$\Rightarrow$ (∗ $\sigma = \delta\cdot\rho$ ∗)

$p \overset{\delta\cdot\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ definition $\overset{\delta\cdot\rho}{\Longrightarrow}$ and definition of quiescence ∗)

$\exists p_1 :\ p \overset{\epsilon}{\Longrightarrow} p_1 \wedge p_1 \xrightarrow{\delta} p_1 \wedge p_1 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ equation (A.2) ∗)

$\exists p_1 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1 \wedge p_1 \xrightarrow{\delta} p_1 \wedge p_1 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ Proposition A.2.2.4 ∗)

$\exists p_1 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \xrightarrow{\delta} \textbf{hide}\, U_2\, \textbf{in}\, p_1 \wedge p_1 \overset{\rho}{\Longrightarrow} p'$

$\Rightarrow$ (∗ induction ∗)

$\exists p_1 :\ \textbf{hide}\, U_2\, \textbf{in}\, p \overset{\epsilon}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \xrightarrow{\delta} \textbf{hide}\, U_2\, \textbf{in}\, p_1$

$\wedge\, \textbf{hide}\, U_2\, \textbf{in}\, p_1 \overset{\rho\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (∗ definition of $\overset{\delta\cdot(\rho\backslash U_2)}{\Longrightarrow}$ ∗)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\delta\cdot(\rho\backslash U_2)}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (∗ Definition A.2.1 ∗)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{(\delta\cdot\rho)\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

$\Rightarrow$ (∗ $\sigma = \delta\cdot\rho$ ∗)

$\textbf{hide}\, U_2\, \textbf{in}\, p \overset{\sigma\backslash U_2}{\Longrightarrow} \textbf{hide}\, U_2\, \textbf{in}\, p'$

2. By induction on the structure of σ (Note that $\sigma \in (I \cup U_1 \cup \{\delta\})^*$):

   $\sigma = \epsilon$:

   The proposition reduces to:

$$\mathbf{hide}\, U_2\, \mathbf{in}\, p \overset{\epsilon}{\Longrightarrow} q \quad \Rightarrow \quad \exists p', \exists \sigma' \in L_\delta^* : \\ p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \wedge \epsilon = \sigma' \backslash U_2 \tag{A.4}$$

   which, using the definition of $\overset{\epsilon}{\Longrightarrow}$, is rewritten to:

$$\mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau^n} q \quad \Rightarrow \quad \exists p', \exists \sigma' \in L_\delta^* : \\ p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \wedge \epsilon = \sigma' \backslash U_2 \tag{A.5}$$

   which is proved by induction on n:

   $n = 0$:

$$\begin{array}{ll} & \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau^n} q \\ \Rightarrow & (*\ n = 0 \text{ and from definition of } \overset{\epsilon}{\Longrightarrow}:\ p \xrightarrow{\tau^0} p' \Leftrightarrow p = p'\ \ *) \\ & \exists p' = p,\ \exists \sigma' = \epsilon :\ p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \wedge \epsilon = \sigma' \backslash U_2 \end{array}$$

   $n = n' + 1$:

$$\begin{array}{ll} & \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau^n} q \\ \Rightarrow & (*\ n = n' + 1 \text{ and definition of } \xrightarrow{\tau^n}\ \ *) \\ & \exists q_1 :\ \mathbf{hide}\, U_2\, \mathbf{in}\, p \xrightarrow{\tau} q_1\ \wedge\ q_1 \xrightarrow{\tau^{n'}} q \\ \Rightarrow & (*\ \text{Proposition A.2.2.6}\ \ *) \\ & \exists q_1 :\ (\ \exists p_1 :\ (\ p \xrightarrow{\tau} p_1 \vee \exists a \in U_2 :\ p \xrightarrow{a} p_1\ ) \\ & \wedge\, q_1 = \mathbf{hide}\, U_2\, \mathbf{in}\, p_1\ )\ \wedge\ q_1 \xrightarrow{\tau^{n'}} q \\ \Rightarrow & (*\ \text{substitution}\ \ *) \\ & \exists p_1 :\ (\ p \xrightarrow{\tau} p_1 \vee \exists a \in U_2 :\ p \xrightarrow{a} p_1\ ) \wedge \mathbf{hide}\, U_2\, \mathbf{in}\, p_1 \xrightarrow{\tau^{n'}} q \\ \Rightarrow & (*\ \text{induction on equation (A.5)}\ \ *) \\ & \exists p_1 :\ (\ p \xrightarrow{\tau} p_1 \vee \exists a \in U_2 :\ p \xrightarrow{a} p_1\ ) \\ & \wedge\, (\ \exists p_2,\ \exists \sigma_1 \in L_\delta^* : \\ & p_1 \overset{\sigma_1}{\Longrightarrow} p_2 \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p_2 \wedge \epsilon = \sigma_1 \backslash U_2\ ) \\ \Rightarrow & (*\ \text{logical manipulation}\ \ *) \\ & (\exists p_1, p_2,\ \exists \sigma_1 \in L_\delta^* : \\ & p \xrightarrow{\tau} p_1 \wedge p_1 \overset{\sigma_1}{\Longrightarrow} p_2 \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p_2 \wedge \epsilon = \sigma_1 \backslash U_2\ ) \\ & \vee\, (\ \exists p_1, p_2,\ \exists \sigma_1 \in L_\delta^*,\ \exists a \in U_2 : \\ & p \xrightarrow{a} p_1 \wedge p_1 \overset{\sigma_1}{\Longrightarrow} p_2 \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p_2 \wedge \epsilon = \sigma_1 \backslash U_2\ ) \\ \Rightarrow & (*\ \text{Definition of } \Longrightarrow\ \ *) \\ & (\exists p' = p_2, \sigma' = \sigma_1 : p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \wedge \epsilon = \sigma' \backslash U_2) \\ & \vee\, (\exists p' = p_2, \sigma' = a {\cdot} \sigma_1 : p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \mathbf{hide}\, U_2\, \mathbf{in}\, p' \wedge \epsilon = \sigma' \backslash U_2) \end{array}$$

$\sigma = a{\cdot}\rho$, with $a \in I \cup U_1$:

$$\begin{array}{ll}
& \textbf{hide}\, U_2 \,\textbf{in}\, p \overset{\sigma}{\Longrightarrow} q \\
\Rightarrow & (*\ \ \sigma = a{\cdot}\rho\ \ *) \\
& \textbf{hide}\, U_2 \,\textbf{in}\, p \overset{a\cdot\rho}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{ definition of } \overset{a\cdot\rho}{\Longrightarrow}\ \ *) \\
& \exists q_1, q_2 :\ \textbf{hide}\, U_2 \,\textbf{in}\, p \overset{\epsilon}{\Longrightarrow} q_1 \wedge q_1 \xrightarrow{a} q_2 \wedge q_2 \overset{\rho}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{ equation (A.4) }\ *) \\
& \exists q_1, q_2 :\ (\ \exists p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \overset{\sigma_1}{\Longrightarrow} p_1 \wedge q_1 = \textbf{hide}\, U_2 \,\textbf{in}\, p_1 \wedge \epsilon = \sigma_1 \backslash U_2\ ) \\
& \wedge\, q_1 \xrightarrow{a} q_2\ \wedge\ q_2 \overset{\rho}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{ substitution }\ *) \\
& \exists q_2, p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \overset{\sigma_1}{\Longrightarrow} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge \textbf{hide}\, U_2 \,\textbf{in}\, p_1 \xrightarrow{a} q_2 \wedge q_2 \overset{\rho}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{ Proposition A.2.2.5 }\ *) \\
& \exists q_2, p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \overset{\sigma_1}{\Longrightarrow} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \\
& \wedge\, (\ \exists p_2 :\ p_1 \xrightarrow{a} p_2 \wedge q_2 = \textbf{hide}\, U_2 \,\textbf{in}\, p_2\ )\ \wedge\ q_2 \overset{\rho}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{ substitution }\ *) \\
& \exists p_1, p_2,\ \exists \sigma_1 \in L_\delta^* : \\
& p \overset{\sigma_1}{\Longrightarrow} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge p_1 \xrightarrow{a} p_2 \wedge \textbf{hide}\, U_2 \,\textbf{in}\, p_2 \overset{\rho}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{ induction }\ *) \\
& \exists p_1, p_2,\ \exists \sigma_1 \in L_\delta^* : \\
& p \overset{\sigma_1}{\Longrightarrow} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge p_1 \xrightarrow{a} p_2 \\
& \wedge\, (\ \exists p_3,\ \exists \sigma_3 \in L_\delta^* : \\
& p_2 \overset{\sigma_3}{\Longrightarrow} p_3 \wedge q = \textbf{hide}\, U_2 \,\textbf{in}\, p_3 \wedge \rho = \sigma_3 \backslash U_2\ ) \\
\Rightarrow & (*\ \text{ logical manipulation }\ *) \\
& \exists p_1, p_2, p_3,\ \exists \sigma_1, \sigma_3 \in L_\delta^* : \\
& p \overset{\sigma_1}{\Longrightarrow} p_1 \wedge p_1 \xrightarrow{a} p_2 \wedge p_2 \overset{\sigma_3}{\Longrightarrow} p_3 \\
& \wedge\, q = \textbf{hide}\, U_2 \,\textbf{in}\, p_3 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge \rho = \sigma_3 \backslash U_2 \\
\Rightarrow & (*\ \ (\sigma_1{\cdot}a{\cdot}\sigma_3)\backslash U_2 = (\sigma_1\backslash U_2){\cdot}(a\backslash U_2){\cdot}(\sigma_3\backslash U_2) = \epsilon{\cdot}a{\cdot}\rho = \sigma\ \ *) \\
& \exists p' = p_3, \sigma' = \sigma_1{\cdot}a{\cdot}\sigma_3 : \\
& p \overset{\sigma'}{\Longrightarrow} p' \wedge q = \textbf{hide}\, U_2 \,\textbf{in}\, p' \wedge \sigma = \sigma' \backslash U_2
\end{array}$$

$\sigma = \delta{\cdot}\rho$:

$$\begin{array}{ll}
& \mathbf{hide}\, U_2 \,\mathbf{in}\, p \xRightarrow{\sigma} q \\
\Rightarrow & (*\ \ \sigma = \delta{\cdot}\rho\ \ *) \\
& \mathbf{hide}\, U_2 \,\mathbf{in}\, p \xRightarrow{\delta\cdot\rho} q \\
\Rightarrow & (*\ \ \text{definition of } \xRightarrow{\delta\cdot\rho}\ \ *) \\
& \exists q_1, q_2 :\ \mathbf{hide}\, U_2 \,\mathbf{in}\, p \xRightarrow{\epsilon} q_1 \wedge q_1 \xrightarrow{\delta} q_2 \wedge q_2 \xRightarrow{\rho} q \\
\Rightarrow & (*\ \ \text{equation (A.4)}\ \ *) \\
& \exists q_1, q_2 :\ (\ \exists p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \xRightarrow{\sigma_1} p_1 \wedge q_1 = \mathbf{hide}\, U_2 \,\mathbf{in}\, p_1 \wedge \epsilon = \sigma_1 \backslash U_2\ ) \\
& \wedge\, q_1 \xrightarrow{\delta} q_2 \wedge q_2 \xRightarrow{\rho} q \\
\Rightarrow & (*\ \ \text{substitution}\ \ *) \\
& \exists q_2, p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \xRightarrow{\sigma_1} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge \mathbf{hide}\, U_2 \,\mathbf{in}\, p_1 \xrightarrow{\delta} q_2 \wedge q_2 \xRightarrow{\rho} q \\
\Rightarrow & (*\ \ \text{Proposition A.2.2.7}\ \ *) \\
& \exists q_2, p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \xRightarrow{\sigma_1} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge p_1 \xrightarrow{\delta} p_1 \\
& \wedge\, q_2 = \mathbf{hide}\, U_2 \,\mathbf{in}\, p_1 \wedge q_2 \xRightarrow{\rho} q \\
\Rightarrow & (*\ \ \text{substitution}\ \ *) \\
& \exists p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \xRightarrow{\sigma_1} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge p_1 \xrightarrow{\delta} p_1 \wedge \mathbf{hide}\, U_2 \,\mathbf{in}\, p_1 \xRightarrow{\rho} q \\
\Rightarrow & (*\ \ \text{induction}\ \ *) \\
& \exists p_1,\ \exists \sigma_1 \in L_\delta^* : \\
& p \xRightarrow{\sigma_1} p_1 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge p_1 \xrightarrow{\delta} p_1 \\
& \wedge\, (\ \exists p_3,\ \exists \sigma_3 \in L_\delta^* : \\
& p_1 \xRightarrow{\sigma_3} p_3 \wedge q = \mathbf{hide}\, U_2 \,\mathbf{in}\, p_3 \wedge \rho = \sigma_3 \backslash U_2\ ) \\
\Rightarrow & (*\ \ \text{logical manipulation}\ \ *) \\
& \exists p_1, p_3,\ \exists \sigma_1, \sigma_3 \in L_\delta^* : \\
& p \xRightarrow{\sigma_1} p_1 \wedge p_1 \xrightarrow{\delta} p_1 \wedge p_1 \xRightarrow{\sigma_3} p_3 \\
& \wedge\, q = \mathbf{hide}\, U_2 \,\mathbf{in}\, p_3 \wedge \epsilon = \sigma_1 \backslash U_2 \wedge \rho = \sigma_3 \backslash U_2 \\
\Rightarrow & (*\ \ (\sigma_1{\cdot}\delta{\cdot}\sigma_3)\backslash U_2 = (\sigma_1 \backslash U_2){\cdot}(\delta \backslash U_2){\cdot}(\sigma_3 \backslash U_2) = \epsilon{\cdot}\delta{\cdot}\rho = \sigma\ \ *) \\
& \exists p' = p_3, \sigma' = \sigma_1{\cdot}\delta{\cdot}\sigma_3 : \\
& p \xRightarrow{\sigma'} p' \wedge q = \mathbf{hide}\, U_2 \,\mathbf{in}\, p' \wedge \sigma = \sigma' \backslash U_2
\end{array}$$

□

Theorem 3.3.5 If $i, s \in \mathbf{IOTS}(I, U)$ with $U_2 \subseteq U$, then:

$$i\ \ \mathbf{ioco}\ \ s\ \Rightarrow\ \mathbf{hide}\, U_2 \,\mathbf{in}\, i\ \ \mathbf{ioco}\ \ \mathbf{hide}\, U_2 \,\mathbf{in}\, s$$

□

Proof To be proved according to Lemma A.1.3:

$$\mathit{Straces}(i) \subseteq \mathit{Straces}(s)\ \Rightarrow\ \mathit{Straces}(\mathbf{hide}\, U_2 \,\mathbf{in}\, i) \subseteq \mathit{Straces}(\mathbf{hide}\, U_2 \,\mathbf{in}\, s)$$

which is straightforward (recall that $L = I \cup U$).

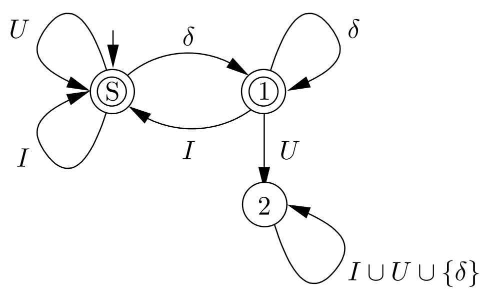


Figure A.1: Automaton that accepts the regular set $(U \cup \delta^* I)^* \delta^*$

$$
\begin{array}{ll}
 & \sigma \in \mathit{Straces}(\mathbf{hide}\, U_2\, \mathbf{in}\, i) \\
\Rightarrow & (*\ \text{Definition } \mathit{Straces}\ *) \\
 & \mathbf{hide}\, U_2\, \mathbf{in}\, i \overset{\sigma}{\Longrightarrow} \\
\Rightarrow & (*\ \text{Proposition A.2.3.2}\ *) \\
 & \exists \sigma' \in L_\delta^* :\ i \overset{\sigma'}{\Longrightarrow} \wedge \sigma = \sigma' \backslash U_2 \\
\Rightarrow & (*\ \text{premise}\ *) \\
 & \exists \sigma' \in L_\delta^* :\ s \overset{\sigma'}{\Longrightarrow} \wedge \sigma = \sigma' \backslash U_2 \\
\Rightarrow & (*\ \text{Proposition A.2.3.1}\ *) \\
 & \exists \sigma' \in L_\delta^* :\ \mathbf{hide}\, U_2\, \mathbf{in}\, s \xRightarrow{\sigma' \backslash U_2} \wedge \sigma = \sigma' \backslash U_2 \\
\Rightarrow & (*\ \text{substitution}\ *) \\
 & \mathbf{hide}\, U_2\, \mathbf{in}\, s \overset{\sigma}{\Longrightarrow} \\
\Rightarrow & (*\ \text{Definition } \mathit{Straces}\ *) \\
 & \sigma \in \mathit{Straces}(\mathbf{hide}\, U_2\, \mathbf{in}\, s)
\end{array}
$$

□

A.3 Proofs of Section 3.4: Underspecification

To improve the readability of the propositions and proofs in this chapter, we make a notational distinction between transitions of s and $\Xi(s)$. For transitions of the IOTS $\Xi(s)$, we use the subscript Ξ. For example, we use $q \xrightarrow{\lambda}_\Xi q'$ to denote $(q, \lambda, q') \in T_{\Xi(s)}$. Likewise we use $\Rightarrow_\Xi$ for sequences of transitions. For transitions and sequences of transitions in s we use the standard notation without subscript.

Lemma A.3.1 $(U \cup \delta^* I)^* \delta^*$ is suffix-closed, i.e., let $\sigma \in (U \cup \delta^* I)^* \delta^*$.

$$\forall (\sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2) : \sigma_2 \in (U \cup \delta^* I)^* \delta^*$$

□

Proof In Figure A.1 we show the automaton that accepts strings in the regular set $(U \cup \delta^* I)^* \delta^*$. The two accepting states are "S" (the starting state) and "1". It is impossible to reach an accepting state from state "2".

So after σ_1 the automaton is in state "S" or "1". In "S", being the start state, the entire regular set is accepted. In state "1" strings of the following form are accepted. Both are subsets of $(U \cup \delta^* I)^* \delta^*$.

1. $\delta^* \subseteq (U \cup \delta^* I)^* \delta^*$
2. $\delta^* I (U \cup \delta^* I)^* \delta^* \subseteq (U \cup \delta^* I)^* \delta^*$

□

The following proposition expresses that Straces do not have outputs after delta.

Proposition A.3.2 Let $s \in \mathbf{LTS}(I, U)$

$$\mathit{Straces}(s) \subseteq (U \cup \delta^* I)^* \delta^*$$

□

Proof The complement of $(U \cup \delta^* I)^* \delta^*$ with respect to L_δ^* are traces of the form $(U \cup \delta^* I)^* \delta^* (\delta U) L_\delta^*$. It is straightforward to verify that this is really the complement by interchanging the accepting and non accepting states in Figure A.1 on the previous page. The result is an automaton that accepts the complement. So a trace in the complement is a trace that contains the sequence $\delta \cdot \lambda$ with $\lambda \in U$. Because of the semantics of quiescence this is of course an impossible trace for an LTS.
Proof by reductio ad absurdum. Suppose the proposition is not true. Let $\sigma = \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \in (U \cup \delta^* I)^* \delta^* (\delta U) L_\delta^*$ where $\sigma_1 \in (U \cup \delta^* I)^* \delta^*, \sigma_2 \in \delta U, \sigma_3 \in L_\delta^*$. Take an arbitrary s with $\sigma \in \mathit{Straces}(s)$, such that $s \xRightarrow{\sigma_1} q_1 \xRightarrow{\sigma_2 \cdot \sigma_3}$

$$\begin{array}{ll}
 & \sigma \in \mathit{Straces}(s) \wedge s \xRightarrow{\sigma_1} q_1 \xrightarrow{\delta} q_2 \\
\Rightarrow & (*\ \text{Definition } \delta\ *) \\
 & \sigma \in \mathit{Straces}(s) \wedge s \xRightarrow{\sigma_1} q_1 \wedge \forall \lambda \in U : q_1 \overset{\lambda}{\not\rightarrow} \wedge q_1 \overset{\tau}{\not\rightarrow} \wedge q_1 = q_2 \\
\Rightarrow & (*\ \text{Definition } \Longrightarrow \text{ and } \sigma_2 \in \delta U\ \ *) \\
 & \sigma \in \mathit{Straces}(s) \wedge s \xRightarrow{\sigma_1} q_1 \wedge q_1 \overset{\sigma_2}{\not\Rightarrow} \\
\Rightarrow & \text{Contradiction}
\end{array}$$

□

Proposition 3.4.5

$$\forall \sigma \in L_\delta^*, q' \in Q_s : s \xRightarrow{\sigma} q' \Leftrightarrow \Xi(s) \xRightarrow{\sigma}_{\Xi} q'$$

□

Proof The crux of the proof is that $\Xi(s)$ only adds states and transitions to s, it does not remove them.

Only if: Proof by induction on the structure of σ.

Basic step: $\sigma = \epsilon$. We prove the following stronger statement by induction on n.

$$\forall q, q' \in Q_s : q \xrightarrow{\tau^n} q' \Rightarrow q \xrightarrow{\tau^n}_{\Xi} q' \tag{A.6}$$

Basic step: $n = 0$

$$\begin{array}{ll} & q \xrightarrow{\tau^0} q' \\ \Rightarrow & (*\ q = q' \text{ and } q, q' \in Q_{\Xi(s)}\ \ *) \\ & q \xrightarrow{\tau^0}_{\Xi} q' \end{array}$$

Induction step: $n = n' + 1$. We assume that equation A.6 holds for n'.

$$\begin{array}{ll} & q \xrightarrow{\tau^{n'}\cdot\tau} q' \\ \Rightarrow & (*\ \text{Definition } \longrightarrow\ \ *) \\ & \exists q_1 \in Q_s : q \xrightarrow{\tau^{n'}} q_1 \wedge q_1 \xrightarrow{\tau} q' \\ \Rightarrow & (*\ \text{Induction}\ \ *) \\ & \exists q_1 \in Q_s : q \xrightarrow{\tau^{n'}}_{\Xi} q_1 \wedge q_1 \xrightarrow{\tau} q' \\ \Rightarrow & (*\ \text{Definition of } \Xi \text{ (note that } (q_1, \tau, q') \in T_s \subseteq T_{\Xi(s)})\ \ *) \\ & \exists q_1 \in Q_s : q \xrightarrow{\tau^{n'}}_{\Xi} q_1 \wedge q_1 \xrightarrow{\tau}_{\Xi} q' \\ \Rightarrow & (*\ \text{Definition } \longrightarrow\ \ *) \\ & q \xrightarrow{\tau^{n'+1}}_{\Xi} q' \end{array}$$

Induction step: $\sigma = \sigma'{\cdot}a$, where $a \in L_\delta$. We assume that the proposition holds for σ'. We identify two cases:

1. $a \in L$

$$\begin{array}{ll} & s \xRightarrow{\sigma'\cdot a} q' \\ \Rightarrow & (*\ \text{Definition } \Longrightarrow\ \ *) \\ & \exists q_1, q_2 \in Q_s : s \xRightarrow{\sigma'} q_1 \wedge q_1 \xrightarrow{a} q_2 \wedge q_2 \xRightarrow{\epsilon} q' \\ \Rightarrow & (*\ \text{Induction}\ \ *) \\ & \exists q_1, q_2 \in Q_s : \Xi(s) \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{a} q_2 \wedge q_2 \xRightarrow{\epsilon} q' \\ \Rightarrow & (*\ \text{Definition of } \Xi(s) \text{ (note that } (q_1, a, q_2) \in T_s \subseteq T_{\Xi(s)})\ \ *) \\ & \exists q_1, q_2 \in Q_s : \Xi(s) \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{a}_{\Xi} q_2 \wedge q_2 \xRightarrow{\epsilon} q' \\ \Rightarrow & (*\ \text{Equation A.6}\ \ *) \\ & \exists q_1, q_2 \in Q_s : \Xi(s) \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{a}_{\Xi} q_2 \wedge q_2 \xRightarrow{\epsilon}_{\Xi} q' \\ \Rightarrow & (*\ \text{Definition } \Longrightarrow\ \ *) \\ & \Xi(s) \xRightarrow{\sigma'\cdot a}_{\Xi} q' \end{array}$$

2. $a = \delta$

$$\begin{array}{ll}
 & s \xRightarrow{\sigma'\cdot\delta} q' \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists q_1, q_2 \in Q_s : s \xRightarrow{\sigma'} q_1 \wedge q_1 \xrightarrow{\delta} q_2 \wedge q_2 \xRightarrow{\epsilon} q' \\
\Rightarrow & (*\ \text{Definition } \delta\ (q_1 = q_2 = q')\ *) \\
 & s \xRightarrow{\sigma'} q' \wedge \forall \lambda \in U_\tau : q' \not\xrightarrow{\lambda} \\
\Rightarrow & (*\ \text{Induction, note that } q' \in Q_s\ *) \\
 & \Xi(s) \xRightarrow{\sigma'}_{\Xi} q' \wedge \forall \lambda \in U_\tau : q' \not\xrightarrow{\lambda} \\
\Rightarrow & (*\ \text{Definition } \Xi\text{, note that } \lambda \notin I \text{ and } q' \in Q_s\ *) \\
 & \Xi(s) \xRightarrow{\sigma'}_{\Xi} q' \wedge \forall \lambda \in U_\tau : q' \not\xrightarrow{\lambda}_{\Xi} \\
\Rightarrow & (*\ \text{Definition } \delta\ *) \\
 & \Xi(s) \xRightarrow{\sigma'}_{\Xi} q' \wedge q' \xrightarrow{\delta}_{\Xi} q' \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \Xi(s) \xRightarrow{\sigma'\cdot\delta}_{\Xi} q'
\end{array}$$

if: Proof by induction on the structure of σ.

Basic step: $\sigma = \epsilon$ We prove the following stronger statement

$$\forall q \in Q_{\Xi(s)}, q' \in Q_s : q \xrightarrow{\tau^n}_{\Xi} q' \Rightarrow q \xrightarrow{\tau^n} q' \qquad \text{(A.7)}$$

which is proven by induction on n:

Basic step: $n = 0$

$$\begin{array}{ll}
 & q \xrightarrow{\tau^0}_{\Xi} q' \\
\Rightarrow & (*\ q = q' \text{ and } q' \in Q_s\ *) \\
 & q \xrightarrow{\tau^0} q'
\end{array}$$

Induction step: $n = n' + 1$. We assume that equation A.7 holds for n'.

$$\begin{array}{ll}
 & q \xrightarrow{\tau^{n'+1}}_{\Xi} q' \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists q_1 \in Q_{\Xi(s)} : q \xrightarrow{\tau^{n'}}_{\Xi} q_1 \wedge q_1 \xrightarrow{\tau}_{\Xi} q' \\
\Rightarrow & (*\ \text{Definition } \Xi\text{, note that } q' \in Q_s\ *) \\
 & \exists q_1 \in Q_s : q \xrightarrow{\tau^{n'}}_{\Xi} q_1 \wedge q_1 \xrightarrow{\tau} q' \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists q_1 \in Q_s : q \xrightarrow{\tau^{n'}} q_1 \wedge q_1 \xrightarrow{\tau} q' \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & q \xrightarrow{\tau^{n'+1}} q'
\end{array}$$

Induction step: $\sigma = \sigma' \cdot a$, where $a \in L_\delta$. We assume that the proposition holds for σ'. We identify two cases:

1. $a \in L$

$\Xi(s) \xRightarrow{\sigma'\cdot a}_{\Xi} q'$

$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)

$\exists q_1, q_2 \in Q_{\Xi(s)} : \Xi(s) \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{a}_{\Xi} q_2 \wedge q_2 \xRightarrow{\epsilon}_{\Xi} q'$

$\Rightarrow$ (∗ Equation A.7, note that $q' \in Q_s$ ∗)

$\exists q_1 \in Q_{\Xi(s)}, q_2 \in Q_s : \Xi(s) \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{a}_{\Xi} q_2 \wedge q_2 \xRightarrow{\epsilon} q'$

$\Rightarrow$ (∗ Definition Ξ, note that $q_2 \in Q_s$ ∗)

$\exists q_1, q_2 \in Q_s : \Xi(s) \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{a} q_2 \wedge q_2 \xRightarrow{\epsilon} q'$

$\Rightarrow$ (∗ Induction ∗)

$\exists q_1, q_2 \in Q_s : s \xRightarrow{\sigma'} q_1 \wedge q_1 \xrightarrow{a} q_2 \wedge q_2 \xRightarrow{\epsilon} q'$

$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)

$s \xRightarrow{\sigma'\cdot a} q'$

2. $a = \delta$

$\Xi(s) \xRightarrow{\sigma'\cdot\delta}_{\Xi} q'$

$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)

$\exists q_1, q_2 \in Q_{\Xi(s)} : s \xRightarrow{\sigma'}_{\Xi} q_1 \wedge q_1 \xrightarrow{\delta}_{\Xi} q_2 \wedge q_2 \xRightarrow{\epsilon}_{\Xi} q'$

$\Rightarrow$ (∗ Definition of δ $(q_1 = q_2 = q')$ ∗)

$\Xi(s) \xRightarrow{\sigma'}_{\Xi} q' \wedge \forall \lambda \in U_\tau : q' \not\xrightarrow{\lambda}_{\Xi}$

$\Rightarrow$ (∗ Induction ∗)

$s \xRightarrow{\sigma'} q' \wedge \forall \lambda \in U_\tau : q' \not\xrightarrow{\lambda}_{\Xi}$

$\Rightarrow$ (∗ $T_s \subseteq T_{\Xi(s)}$ ∗)

$s \xRightarrow{\sigma'} q' \wedge \forall \lambda \in U_\tau : q' \not\xrightarrow{\lambda}$

$\Rightarrow$ (∗ Definition δ ∗)

$s \xRightarrow{\sigma'} q' \wedge q' \xrightarrow{\delta} q'$

$\Rightarrow$ (∗ Definition of $\Longrightarrow$ ∗)

$s \xRightarrow{\sigma'\cdot\delta} q'$

□

Lemma A.3.3 Let q_χ be the chaotic state in a demonically completed LTS:

$$\forall \sigma \in (U \cup \delta^* I)^* : q_\chi \xRightarrow{\sigma}_{\Xi} q_\chi$$

□

Proof Proof by induction on the structure of σ.

Basic step $\sigma = \epsilon$:

Always by definition: $q_\chi \xrightarrow{\tau^0}_{\Xi} q_\chi$

Induction step : Suppose the lemma holds for σ'. We prove that it holds for $\sigma \in \sigma' \cdot (U \cup \delta^* I)$.

1. $\sigma = \sigma'\cdot\lambda$ where $\lambda \in U$
   $q_\chi \xRightarrow{\sigma'}_\Xi q_\chi$
   $\Rightarrow$ (* definition Ξ : $\forall \lambda \in U : q_\chi \xrightarrow{\tau}_\Xi q_\Omega \xrightarrow{\lambda}_\Xi q_\chi$ *)
   $q_\chi \xRightarrow{\sigma'}_\Xi q_\chi \xRightarrow{\lambda}_\Xi q_\chi$
   $\Rightarrow$ (* Definition $\Longrightarrow$ *)
   $q_\chi \xRightarrow{\sigma'\cdot\lambda}_\Xi q_\chi$
   $\Rightarrow$ (* $\sigma = \sigma'\cdot\lambda$ *)
   $q_\chi \xRightarrow{\sigma}_\Xi q_\chi$
2. $\sigma \in \sigma'\cdot\delta^*\cdot\lambda$ where $\lambda \in I$
   $q_\chi \xRightarrow{\sigma'}_\Xi q_\chi$
   $\Rightarrow$ (* definition Ξ : $\forall \lambda \in I : q_\chi \xrightarrow{\tau}_\Xi q_\Delta \xrightarrow{\delta^*}_\Xi q_\Delta \xrightarrow{\lambda}_\Xi q_\chi$ *)
   $q_\chi \xRightarrow{\sigma'}_\Xi q_\chi \xRightarrow{\delta^*\cdot\lambda}_\Xi q_\chi$
   $\Rightarrow$ (* Definition $\Longrightarrow$ *)
   $q_\chi \xRightarrow{\sigma'\cdot\delta^*\cdot\lambda}_\Xi q_\chi$
   $\Rightarrow$ (* $\sigma \in \sigma'\cdot\delta^*\cdot\lambda$ *)
   $q_\chi \xRightarrow{\sigma}_\Xi q_\chi$

□

Proposition A.3.4 Let q_χ be the chaotic state in a demonically completed LTS:

$$\forall q' \in Q_{\Xi(s)}, \sigma \in (U \cup \delta^* I)^*\delta^* : q_\chi \xRightarrow{\sigma}_\Xi q' \Rightarrow q' \notin Q_s$$

□

Proof From Lemma A.3.3 we already know that the lemma holds for traces in the regular set $(U \cup \delta^* I)^*$. Now we have to prove that the lemma holds if traces of this form end with δ^*. Let $\sigma \in (U \cup \delta^* I)^*$

$\sigma \in (U \cup \delta^* I)^*$
$\Rightarrow$ (* Lemma A.3.3 *)
$q_\chi \xRightarrow{\sigma}_\Xi q_\chi$
$\Rightarrow$ (* Definition Ξ *)
$q_\chi \xRightarrow{\sigma}_\Xi q_\chi \xrightarrow{\tau}_\Xi q_\Delta \xrightarrow{\delta^*}_\Xi q_\Delta$
$\Rightarrow$ (* Definition $\Rightarrow$ *)
$q_\chi \xRightarrow{\sigma\cdot\delta^*}_\Xi q_\Delta$

□

We delay the proof of Theorem 3.4.6 until after Theorem 3.4.10.

Proposition A.3.5 Let $s \in \mathbf{LTS}(I, U)$.

$$\mathit{Utraces}(s) = \{\sigma \in \mathit{Straces}(s) \mid (\Xi(s)\ \mathbf{after}\ \sigma) \subseteq Q_s\}$$

□

Proof We prove the proposition in two steps:

1. $\mathit{Utraces}(s) \subseteq \{\sigma \in \mathit{Straces}(s) \mid (\Xi(s)\ \textbf{after}\ \sigma) \subseteq Q_s\}$

   We will prove this equation by reductio ad absurdum. Suppose the above proposition does not hold, then there exists a σ such that the following holds:
   $\sigma \in \mathit{Utraces}(s) \wedge (\sigma \notin \mathit{Straces}(s) \vee (\Xi(s)\ \textbf{after}\ \sigma) \not\subseteq Q_s)$
   There are two possibilities:

   (a) $\sigma \in \mathit{Utraces}(s) \wedge \sigma \notin \mathit{Straces}(s)$. We prove that this gives rise to a contradiction.
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \sigma \notin \mathit{Straces}(s)$
   $\Rightarrow$ $(*$ Definition $\mathit{Straces}$ $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge (\sigma \notin L_\delta^* \vee s \overset{\sigma}{\not\Longrightarrow})$
   $\Rightarrow$ $(*$ Definition $\mathit{Utraces}$ $*)$
   $\quad \sigma \in L_\delta^* \wedge s \overset{\sigma}{\Longrightarrow} \wedge (\sigma \notin L_\delta^* \vee s \overset{\sigma}{\not\Longrightarrow})$
   $\Rightarrow$ Contradiction

   (b) $\sigma \in \mathit{Utraces}(s) \wedge (\Xi(s)\ \textbf{after}\ \sigma) \not\subseteq Q_s$. We prove that this gives also rise to a contradiction.
   $\quad \sigma \in \mathit{Utraces}(s) \wedge (\Xi(s)\ \textbf{after}\ \sigma) \not\subseteq Q_s$
   $\Rightarrow$ $(*$ Definition **after** $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \{q \mid \Xi(s) \overset{\sigma}{\Longrightarrow}_\Xi q\} \not\subseteq Q_s$
   $\Rightarrow$ $(*$ Set definition $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \exists q \notin Q_s : \Xi(s) \overset{\sigma}{\Longrightarrow}_\Xi q$
   $\Rightarrow$ $(*$ Definition Ξ, take the first time that $\Xi(s)$ enters a state not in Q_s. This is always via an input action. $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \exists q \notin Q_s, q' \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I :$
   $\quad \sigma_1 {\cdot} \lambda {\cdot} \sigma_2 = \sigma \wedge \Xi(s) \overset{\sigma_1}{\Longrightarrow}_\Xi q' \wedge q' \xrightarrow{\lambda}_\Xi q_\chi \overset{\sigma_2}{\Longrightarrow}_\Xi q$
   $\Rightarrow$ $(*$ Proposition 3.4.5 $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \exists q' \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I : \sigma_1 {\cdot} \lambda {\cdot} \sigma_2 = \sigma$
   $\quad \wedge\, s \overset{\sigma_1}{\Longrightarrow} q' \wedge q' \xrightarrow{\lambda}_\Xi q_\chi$
   $\Rightarrow$ $(*$ Definition Ξ $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \exists q' \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I : \sigma_1 {\cdot} \lambda {\cdot} \sigma_2 = \sigma$
   $\quad \wedge\, s \overset{\sigma_1}{\Longrightarrow} q' \wedge q' \overset{\lambda}{\not\longrightarrow} \wedge\, q' \overset{\tau}{\not\longrightarrow}$
   $\Rightarrow$ $(*$ Definition $\overset{\lambda}{\Longrightarrow}$ $*)$
   $\quad \sigma \in \mathit{Utraces}(s) \wedge \exists q' \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I : \sigma_1 {\cdot} \lambda {\cdot} \sigma_2 = \sigma$
   $\quad \wedge\, s \overset{\sigma_1}{\Longrightarrow} q' \wedge q' \overset{\lambda}{\not\Longrightarrow}$
   $\Rightarrow$ $(*$ Definition $\mathit{Utraces}$ $*)$
   $\quad (\sigma \in L_\delta^* \wedge s \overset{\sigma}{\Longrightarrow} \wedge \nexists q' \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I : \sigma_1 {\cdot} \lambda {\cdot} \sigma_2 = \sigma$
   $\quad \wedge\, s \overset{\sigma_1}{\Longrightarrow} q' \wedge q' \overset{\lambda}{\not\Longrightarrow})$
   $\quad \wedge\, (\exists q' \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I : \sigma_1 {\cdot} \lambda {\cdot} \sigma_2 = \sigma \wedge s \overset{\sigma_1}{\Longrightarrow} q' \wedge q' \overset{\lambda}{\not\Longrightarrow})$
   $\Rightarrow$ Contradiction

2. $\mathit{Utraces}(s) \supseteq \{\sigma \in \mathit{Straces}(s) \mid (\Xi(s) \textbf{ after } \sigma) \subseteq Q_s\}$.

   We will prove this equation by reductio ad absurdum. Suppose that the above proposition does not hold, then the following holds: There exists a trace σ such that

$$\begin{array}{ll}
 & \sigma \in \mathit{Straces}(s) \wedge \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge \sigma \notin \mathit{Utraces}(s) \\
\Rightarrow & (*\ \text{Definition } \mathit{Utraces}\ *) \\
 & \sigma \in \mathit{Straces}(s) \wedge \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge (\sigma \notin L_\delta^* \vee s \overset{\sigma}{\not\Longrightarrow} \\
 & \vee \exists (q_1, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge s \overset{\sigma_1}{\Longrightarrow} q_1 \wedge q_1 \overset{\lambda}{\not\Longrightarrow}) \\
\Rightarrow & (*\ \text{Definition } \mathit{Straces}\ *) \\
 & \sigma \in L_\delta^* \wedge s \overset{\sigma}{\Longrightarrow} \wedge \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge (\sigma \notin L_\delta^* \vee s \overset{\sigma}{\not\Longrightarrow} \\
 & \vee \exists (q_1 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge s \overset{\sigma_1}{\Longrightarrow} q_1 \wedge q_1 \overset{\lambda}{\not\Longrightarrow})
\end{array}$$

   There are three possibilities in the last conjunct:

   (a) $\sigma \notin L_\delta^*$
   $\Rightarrow$ Contradiction

   (b) $s \overset{\sigma}{\not\Longrightarrow}$
   $\Rightarrow$ Contradiction

   (c) $\exists (q_1 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge s \overset{\sigma_1}{\Longrightarrow} q_1 \wedge q_1 \overset{\lambda}{\not\Longrightarrow}$

$$\begin{array}{ll}
\Rightarrow & (*\ s \text{ is strongly converging}\ *) \\
 & \sigma \in L_\delta^* \wedge s \overset{\sigma}{\Longrightarrow} \wedge \Xi(s) \textbf{ after } \sigma \subseteq Q_s \\
 & \wedge \exists (q_1, q_2 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge s \overset{\sigma_1}{\Longrightarrow} q_1 \\
 & \wedge q_1 \overset{\epsilon}{\Longrightarrow} q_2 \wedge q_2 \overset{\lambda}{\not\longrightarrow} \wedge q_2 \overset{\tau}{\not\longrightarrow} \\
\Rightarrow & (*\ \text{Definition } \overset{\sigma_1}{\Longrightarrow}\ *) \\
 & \Xi(s) \textbf{ after } \sigma \subseteq Q_s \\
 & \wedge \exists (q_2 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge s \overset{\sigma_1}{\Longrightarrow} q_2 \\
 & \wedge q_2 \overset{\lambda}{\not\longrightarrow} \wedge q_2 \overset{\tau}{\not\longrightarrow} \\
\Rightarrow & (*\ \text{Proposition 3.4.5}\ *) \\
 & \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge \exists (q_2 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \\
 & \wedge \Xi(s) \overset{\sigma_1}{\Longrightarrow}_\Xi q_2 \wedge q_2 \overset{\lambda}{\not\longrightarrow} \wedge q_2 \overset{\tau}{\not\longrightarrow} \\
\Rightarrow & (*\ \text{Definition } \Xi\ *) \\
 & \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge \exists (q_2 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \\
 & \wedge \Xi(s) \overset{\sigma_1}{\Longrightarrow}_\Xi q_2 \wedge q_2 \overset{\lambda}{\longrightarrow}_\Xi q_\chi \\
\Rightarrow & (*\ \text{Proposition A.3.2 } (\sigma \in \mathit{Straces}(s))\ *) \\
 & \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge \sigma \in (U \cup \delta^* I)^* \delta^* \\
 & \wedge \exists (q_2 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge \Xi(s) \overset{\sigma_1}{\Longrightarrow}_\Xi q_2 \\
 & \wedge q_2 \overset{\lambda}{\longrightarrow}_\Xi q_\chi \\
\Rightarrow & (*\ \text{Lemma A.3.1}\ *) \\
 & \Xi(s) \textbf{ after } \sigma \subseteq Q_s \wedge \sigma \in (U \cup \delta^* I)^* \delta^* \\
 & \wedge \exists (q_2 \in Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) : \sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge \sigma_2 \in (U \cup \delta^* I)^* \delta^* \\
 & \wedge \Xi(s) \overset{\sigma_1}{\Longrightarrow}_\Xi q_2 \wedge q_2 \overset{\lambda}{\longrightarrow}_\Xi q_\chi
\end{array}$$

$\Rightarrow$ (∗ Proposition A.3.4 ∗)
$\Xi(s)$ **after** $\sigma \subseteq Q_s \wedge \exists (q \notin Q_s, \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in I) :$
$\sigma_1 \cdot \lambda \cdot \sigma_2 = \sigma \wedge \Xi(s) \xRightarrow{\sigma_1}_\Xi q_2 \xrightarrow{\lambda}_\Xi q_\chi \xRightarrow{\sigma_2} q$
$\Rightarrow$ (∗ Definition $\subseteq$ ∗)
$\Xi(s)$ **after** $\sigma \subseteq Q_s \wedge \{q \mid \Xi(s) \xRightarrow{\sigma} q\} \not\subseteq Q_s$
$\Rightarrow$ (∗ Definition **after** ∗)
$\Xi(s)$ **after** $\sigma \subseteq Q_s \wedge \Xi(s)$ **after** $\sigma \not\subseteq Q_s$
$\Rightarrow$ Contradiction

□

Lemma A.3.6 $\forall \sigma \in \mathit{Utraces}(s) : \mathit{out}(s$ **after** $\sigma) = \mathit{out}(\Xi(s)$ **after** $\sigma)$ □

Proof

⊆: We prove $x \in \mathit{out}(s$ **after** $\sigma) \Rightarrow x \in \mathit{out}(\Xi(s)$ **after** $\sigma)$.

$x \in \mathit{out}(s$ **after** $\sigma)$
$\Rightarrow$ (∗ Definitions *out* and **after** ∗)
$\exists q \in Q_s : s \xRightarrow{\sigma \cdot x} q$
$\Rightarrow$ (∗ Proposition 3.4.5 ∗)
$\exists q \in Q_s : \Xi(s) \xRightarrow{\sigma \cdot x}_\Xi q$
$\Rightarrow$ (∗ Definitions *out* and **after** ∗)
$x \in \mathit{out}(\Xi(s)$ **after** $\sigma)$

⊇: We prove $x \in \mathit{out}(\Xi(s)$ **after** $\sigma) \Rightarrow x \in \mathit{out}(s$ **after** $\sigma)$.

$x \in \mathit{out}(\Xi(s)$ **after** $\sigma)$
$\Rightarrow$ (∗ Definition *out* and **after** ∗)
$\exists q' \in Q_{\Xi(s)} : \Xi(s) \xRightarrow{\sigma \cdot x}_\Xi q'$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists q, q' \in Q_{\Xi(s)} : \Xi(s) \xRightarrow{\sigma}_\Xi q \xRightarrow{x}_\Xi q'$
$\Rightarrow$ (∗ Proposition A.3.5, premise $\sigma \in \mathit{Utraces}(s)$ ∗)
$\exists q \in Q_s, q' \in Q_{\Xi(s)} : \Xi(s) \xRightarrow{\sigma}_\Xi q \xRightarrow{x}_\Xi q'$
$\Rightarrow$ (∗ Definition 3.4.3: Ξ only adds transitions for input actions ∗)
$\exists q, q' \in Q_s : \Xi(s) \xRightarrow{\sigma}_\Xi q \xRightarrow{x}_\Xi q'$
$\Rightarrow$ (∗ Proposition 3.4.5 ∗)
$\exists q, q' \in Q_s : s \xRightarrow{\sigma} q \xRightarrow{x} q'$
$\Rightarrow$ (∗ Definitions *out* and **after** ∗)
$x \in \mathit{out}(s$ **after** $\sigma)$

□

Lemma A.3.7 $\forall s \in \mathbf{LTS}(I, U) : \mathit{Utraces}(s) \subseteq \mathit{Straces}(\Xi(s))$

□

Proof

$$\begin{array}{ll} & \sigma \in \mathit{Utraces}(s) \\ \Rightarrow & (*\ \text{Definition } \mathit{Utraces}\ *) \\ & \exists q' : s \overset{\sigma}{\Longrightarrow} q' \\ \Rightarrow & (*\ \text{Proposition 3.4.5}\ *) \\ & \exists q' \in Q_s : \Xi(s) \overset{\sigma}{\Longrightarrow}_{\Xi} q' \\ \Rightarrow & (*\ \text{Definition } \mathit{Straces}\ *) \\ & \sigma \in \mathit{Straces}(\Xi(s)) \end{array}$$

□

Proposition A.3.8 $\sigma \in \mathit{Straces}(\Xi(s))\backslash \mathit{Utraces}(s) \Rightarrow \mathit{out}(\Xi(s)\ \textbf{after}\ \sigma) = U_\delta$ □

Proof Let $\sigma \in \mathit{Straces}(\Xi(s))\backslash \mathit{Utraces}(s)$

$$\begin{array}{ll} & \exists q \in Q_{\Xi(s)} : \Xi(s) \overset{\sigma}{\Longrightarrow}_{\Xi} q \\ \Rightarrow & (*\ \text{Proposition A.3.5}\ *) \\ & \exists q \in Q_{\Xi(s)}\backslash Q_s : \Xi(s) \overset{\sigma}{\Longrightarrow}_{\Xi} q \\ \Rightarrow & (*\ \text{Definition 3.4.3, we are in the demonic process}\ *) \\ & \exists q \in \{q_\chi, q_\Delta, q_\Omega\} : \Xi(s) \overset{\sigma}{\Longrightarrow}_{\Xi} \\ \Rightarrow & (*\ \text{Definition } \mathit{out} \text{ and } \textbf{after}\ *) \\ & \mathit{out}(Xi(s)\ \textbf{after}\ \sigma) = U_\delta \end{array}$$

Rationale: q_Δ can do output δ, q_Ω can do all outputs in U, and q_χ can do outputs in U_δ.

□

Theorem 3.4.10

$$\textbf{uioco} = \ \textbf{ioco} \circ \Xi$$

□

Proof Let $i \in \textbf{IOTS}(I, U), s \in \textbf{LTS}(I, U)$ To make the proof easier to read we first expand the **uioco** and **ioco** $\circ\, \Xi$ definitions:
$\forall \sigma \in \mathit{Utraces}(s) : \mathit{out}(i\ \textbf{after}\ \sigma) \subseteq \mathit{out}(s\ \textbf{after}\ \sigma)$
$\Leftrightarrow \forall \sigma \in \mathit{Straces}(\Xi(s)) : \mathit{out}(i\ \textbf{after}\ \sigma) \subseteq \mathit{out}(\Xi(s)\ \textbf{after}\ \sigma)$

only if: Take $\sigma \in \mathit{Straces}(\Xi(s))$ arbitrary. There are two possibilities:

1. $\sigma \in \mathit{Straces}(\Xi(s)) \cap \mathit{Utraces}(s)$

$$\begin{array}{ll} & \sigma \in \mathit{Utraces}(s) \wedge \mathit{out}(i\ \textbf{after}\ \sigma) \subseteq \mathit{out}(s\ \textbf{after}\ \sigma) \\ \Rightarrow & (*\ \text{Lemma A.3.6}\ *) \\ & \sigma \in \mathit{Utraces}(s) \wedge \mathit{out}(i\ \textbf{after}\ \sigma) \subseteq \mathit{out}(\Xi(s)\ \textbf{after}\ \sigma) \end{array}$$

2. $\sigma \in \mathit{Straces}(\Xi(s))\backslash \mathit{Utraces}(s)$

$$\begin{array}{ll} \Rightarrow & (*\ \text{Proposition A.3.8}\ *) \\ & \mathit{out}(\Xi(s)\ \textbf{after}\ \sigma) = U_\delta \\ \Rightarrow & (*\ \text{Logical reasoning}\ *) \\ & \mathit{out}(i\ \textbf{after}\ \sigma) \subseteq \mathit{out}(\Xi(s)\ \textbf{after}\ \sigma) \end{array}$$

if:

$$\begin{array}{rl} & \forall \sigma \in \mathit{Straces}(\Xi(s)) : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(\Xi(s) \textbf{ after } \sigma) \\ \Rightarrow & (*\ \text{Lemma A.3.7}\ *) \\ & \forall \sigma \in \mathit{Utraces}(s) : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(\Xi(s) \textbf{ after } \sigma) \\ \Rightarrow & (*\ \text{Lemma A.3.6}\ *) \\ & \forall \sigma \in \mathit{Utraces}(s) : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(s \textbf{ after } \sigma) \end{array}$$

□

Theorem 3.4.6 Let $i \in \mathbf{IOTS}(I, U), s \in \mathbf{LTS}(I, U)$ then

$$i \textbf{ ioco } s \Rightarrow i(\textbf{ioco} \circ \Xi)s$$

Proof This theorem follows from Theorem 3.4.10. □

A.4 Proofs of Section 3.4.3: New parallel composition operator

In the proofs we use the notation $r][s$ in two ways. One is to refer to the parallel composition of the transition systems r and s. The other is to refer to the state tuple $(r, s) \in Q_{r][s}$. In the latter case r and s refer to states in Q_r and Q_s. Throughout the proofs, when we compose two systems, say $r \in \mathbf{LTS}(I_r, U_r)$ and $s \in \mathbf{LTS}(I_s, U_s)$, these systems have the following signature, unless stated otherwise: $I_r \cap I_s = U_r \cap U_s = \emptyset$.

For a transition system s, we denote its corresponding chaotic transition system as $\chi(s)$. This means that it is a chaotic process with the label-set of s. We may refer to the states of the chaotic process as $\chi(s)$, $\Delta(s)$ and $\Omega(s)$. Normally it is clear from the context whether we are referring to a chaotic transition system or to its states. We use the notation Q_χ to refer to $\{\chi, \Delta, \Omega\}$, or $Q_{\chi(s)}$ if it is not directly clear which label-sets are involved.

Before we prove Theorem 3.4.11 we define some lemmas to describe the characteristics of traces in parallelly composed systems. The following lemma describes the characteristics of the new parallel composition operator for single actions.

Lemma A.4.1 Let $r \in \mathbf{LTS}(I_r, U_r), s \in \mathbf{LTS}(I_s, U_s)$.

1. $\forall a \in L_r \setminus L_s : r][s \xrightarrow{a} q \Leftrightarrow \exists r' \in Q_r : r \xrightarrow{a} r' \wedge q = r'][s$
2. $\forall a \in L_s \setminus L_r : r][s \xrightarrow{a} q \Leftrightarrow \exists s' \in Q_s : s \xrightarrow{a} s' \wedge q = r][s'$
3. $r][s \xrightarrow{\tau} q \Leftrightarrow$
   $(\exists r' \in Q_r : r \xrightarrow{\tau} r' \wedge q = r'][s) \vee (\exists s' \in Q_s : s \xrightarrow{\tau} s' \wedge q = r][s')$
4. $\forall x \in (U_r \cap I_s) : r][s \xrightarrow{x} q \Leftrightarrow$
   $(\exists r' \in Q_r : r \xrightarrow{x} r' \wedge s \not\xrightarrow{x} \wedge s \not\xrightarrow{\tau} \wedge q = r'][\chi(s))$
   $\vee (\exists r' \in Q_r, s' \in Q_s : r \xrightarrow{x} r' \wedge s \xrightarrow{x} s' \wedge q = r'][s')$

5. $\forall x \in (I_r \cap U_s) : r][s \xrightarrow{x} q \Leftrightarrow$
   $(r \overset{x}{\not\longrightarrow} \wedge r \overset{\tau}{\not\longrightarrow} \wedge \exists s' \in Q_s : s \xrightarrow{x} s' \wedge q = \chi(r)][s')$
   $\vee (\exists r' \in Q_r, s' \in Q_s : r \xrightarrow{x} r' \wedge s \xrightarrow{x} s' \wedge q = r'][s')$

6. $r][s \xrightarrow{\delta} q \Leftrightarrow r \xrightarrow{\delta} r \wedge s \xrightarrow{\delta} s \wedge q = r][s$

$\square$

Proof

1. Let $a \in L_r \setminus L_s$

   Only if:

   $r][s \xrightarrow{a} q$
   $\Rightarrow$ (∗ definition][case 1 ∗)
   $\exists r' \in Q_r : r \xrightarrow{a} r' \wedge q = r'][s$

   If:

   $\exists r' \in Q_r : r \xrightarrow{a} r' \wedge q = r'][s$
   $\Rightarrow$ (∗ definition][case 1 ∗)
   $r][s \xrightarrow{a} q$

2. Symmetrical to 1

3. Analogous to 1

4. The definition of][, cases 3 and 5, gives the following options:

   (a) $s \overset{x}{\not\longrightarrow} \wedge s \overset{\tau}{\not\longrightarrow}$,
   (b) $\exists r' \in Q_r, s' \in Q_s : r \xrightarrow{x} r' \wedge s \xrightarrow{x} s'$.

   We start with the first: $s \overset{x}{\not\longrightarrow} \wedge s \overset{\tau}{\not\longrightarrow}$

   Only if:

   $r][s \xrightarrow{x} q$
   $\Rightarrow$ (∗ premise: $s \overset{x}{\not\longrightarrow} \wedge s \overset{\tau}{\not\longrightarrow}$ ∗)
   $\exists r' \in Q_r : r \xrightarrow{x} r' \wedge s \overset{x}{\not\longrightarrow}$
   $\Rightarrow$ (∗ Definition][case 3 ∗)
   $\exists r' \in Q_r : r \xrightarrow{x} r' \wedge s \overset{x}{\not\longrightarrow} \wedge s \overset{\tau}{\not\longrightarrow} \wedge q = r'][\chi(s)$

   The second case is analogous to the previous case. Note that with this case there may be some confusion about to which system the states r and s belong: it may be the non-chaotic system or the chaotic system. For the proof it does not matter: if $r \in Q_r$ or $r \in Q_{\chi(r)}$; if it is in Q_r we perform transitions from system r and if it is in $Q_{\chi(r)}$ we perform transitions from $\chi(r)$.

If: We continue with the first case.

$$
\begin{array}{ll}
 & \exists r' \in Q_r : r \xrightarrow{x} r' \wedge s \not\xrightarrow{x} \wedge s \not\xrightarrow{\tau} \wedge q = r'][\chi(s) \\
\Rightarrow & (*\ \text{Definition }][\ \text{case 3}\ \ *) \\
 & r][s \xrightarrow{x} r'][\chi(s) \wedge q = r'][\chi(s) \\
\Rightarrow & (*\ \text{Logical reasoning}\ \ *) \\
 & r][s \xrightarrow{x} q
\end{array}
$$

The second case is analogous to the first case.

5. This case is symmetrical to the previous case.

6. $a = \delta$. A state is quiescent if it cannot perform an output action. For a parallel composition this means that none of the components can perform an output action.

Only if:

$$
\begin{array}{ll}
 & r][s \xrightarrow{\delta} q \\
\Rightarrow & (*\ \text{Definition } \delta\ \ *) \\
 & \forall \mu \in U_{r][s} \cup \{\tau\} : r][s \not\xrightarrow{\mu}
\end{array}
$$

Based on the definition of $r][s$ we identify the following output actions ($U_{r][s} = U_r \cup U_s$):

- $x \in U_r \backslash L_s$
- $x \in U_s \backslash L_r$
- $x \in U_r \cap I_s$
- $x \in U_s \cap I_r$

The parallel composition cannot do any of these actions, this means that the individual components also cannot do any of these actions.

$$
\begin{array}{ll}
\Rightarrow & (*\ \text{Definition }][,\ \text{all cases}\ \ *) \\
 & \forall \mu \in (U_r \backslash L_s) \cup \{\tau\} : r \not\xrightarrow{\mu} \\
 & \wedge \forall \mu \in (U_s \backslash L_r) \cup \{\tau\} : s \not\xrightarrow{\mu} \\
 & \wedge \forall \mu \in U_r \cap I_s \cup \{\tau\} : r \not\xrightarrow{\mu} \\
 & \wedge \forall \mu \in U_s \cap I_r \cup \{\tau\} : s \not\xrightarrow{\mu} \\
\Rightarrow & (*\ \text{Logical reasoning. Premise: } I_r \cap I_s = U_r \cap U_s = \emptyset\ \ *) \\
 & \forall \mu \in U_r \cup \{\tau\} : r \not\xrightarrow{\mu} \wedge \forall \mu \in U_s \cup \{\tau\} : s \not\xrightarrow{\mu} \\
\Rightarrow & (*\ \text{definition } \delta\ \ *) \\
 & r \xrightarrow{\delta} r \wedge s \xrightarrow{\delta} s \wedge q = r][s
\end{array}
$$

If:

$$\begin{array}{ll}
 & r \xrightarrow{\delta} r \wedge s \xrightarrow{\delta} s \wedge q = r][s \\
\Rightarrow & (*\ \text{definition } \delta\ *) \\
 & \forall \mu \in U_r \cup \{\tau\} : r \xrightarrow{\mu}\!\!\!\!/\ \ \wedge \forall \mu \in U_s \cup \{\tau\} : s \xrightarrow{\mu}\!\!\!\!/\ \ \wedge q = r][s \\
\Rightarrow & (*\ \text{definition }][\text{, with } I_r \cap I_s = U_r \cap U_s = \emptyset\ *) \\
 & \forall \mu \in U_r \cup U_s \cup \{\tau\} : r][s \xrightarrow{\mu}\!\!\!\!/\ \ \wedge q = r][s \\
\Rightarrow & (*\ \text{definition } \delta\ *) \\
 & r][s \xrightarrow{\delta} r][s \wedge q = r][s \\
\Rightarrow & (*\ \text{Logical reasoning } (q = r][s)\ *) \\
 & r][s \xrightarrow{\delta} q
\end{array}$$

□

In the following lemmas we abstract from τ transitions from composed systems. We start with the following lemma about empty traces in a parallel composition.

Lemma A.4.2 Let $r, s \in$ **LTS**

$$\begin{array}{ll}
r][s \xrightarrow{\tau^n} q \Leftrightarrow & \exists r' \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^i} r' \wedge s \xrightarrow{\tau^j} s' \\
 & \wedge q = r'][s' \wedge n = i + j
\end{array}$$

□

Proof

Only if: Proof by induction on n.

Basic step: $n = 0$

$$\begin{array}{ll}
 & r][s \xrightarrow{\tau^0} q \\
\Rightarrow & (*\ \tau^0 \text{ means we stay in the same state}\ *) \\
 & r][s \xrightarrow{\tau^0} q \wedge q = r][s \\
\Rightarrow & (*\ \tau^0 \text{ means we stay in the same state}\ *) \\
 & q = r][s \wedge r \xrightarrow{\tau^0} r \wedge s \xrightarrow{\tau^0} s
\end{array}$$

Induction step: $n = m + 1$ and assume that the lemma holds for m.

$$\begin{array}{ll}
 & r][s \xrightarrow{\tau^{m+1}} q \\
\Rightarrow & (*\ \text{Definition } \longrightarrow\ *) \\
 & \exists q_1 \in Q_{r][s} : r][s \xrightarrow{\tau^m} q_1 \xrightarrow{\tau} q \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists q_1 \in Q_{r][s} : q_1 \xrightarrow{\tau} q \\
 & \wedge \exists r_1 \in Q_r, s_1 \in Q_s, k, l \geq 0 : r \xrightarrow{\tau^k} r_1 \wedge s \xrightarrow{\tau^l} s_1 \\
 & \wedge q_1 = r_1][s_1 \wedge m = k + l
\end{array}$$

$\Rightarrow$ (∗ Lemma A.4.1 case 3 ∗)
$\exists r_1 \in Q_r, s_1 \in Q_s, k, l \geq 0 : r \xrightarrow{\tau^k} r_1 \wedge s \xrightarrow{\tau^l} s_1 \wedge m = k + l$
$\wedge ((\exists r' \in Q_r : r_1 \xrightarrow{\tau} r' \wedge q = r'][s_1)$
$\vee (\exists s' \in Q_s : s_1 \xrightarrow{\tau} s' \wedge q = r_1][s'))$
$\Rightarrow$ (∗ Logical reasoning ∗)
$((\exists r_1, r' \in Q_r, s_1 \in Q_s, k, l \geq 0 : r \xrightarrow{\tau^k} r_1 \xrightarrow{\tau} r' \wedge s \xrightarrow{\tau^l} s_1)$
$\vee (\exists r_1 \in Q_r, s_1, s' \in Q_s, k, l \geq 0 : r \xrightarrow{\tau^k} r_1 \wedge s \xrightarrow{\tau^l} s_1 \xrightarrow{\tau} s'))$
$\wedge m = k + l$
$\Rightarrow$ (∗ Definition $\longrightarrow$ ∗)
$((\exists r' \in Q_r, s_1 \in Q_s, k, l \geq 0 : r \xrightarrow{\tau^{k+1}} r' \wedge s \xrightarrow{\tau^l} s_1)$
$\vee (\exists r_1 \in Q_r, s' \in Q_s, k, l \geq 0 : r \xrightarrow{\tau^k} r_1 \wedge s \xrightarrow{\tau^{l+1}} s'))$
$\wedge m = k + l$
$\Rightarrow$ (∗ Logical reasoning: $n = m + 1$ and $m = k + l$ ∗)
$\exists r' \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^i} r' \wedge s \xrightarrow{\tau^j} s' \wedge q = r'][q'$
$\wedge n = i + j$

If: Proof by induction on n.

Basic step: $n = 0$

$\exists r' \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^i} r' \wedge s \xrightarrow{\tau^j} s' \wedge q = r'][s'$
$\wedge i + j = 0$
$\Rightarrow$ (∗ Logical reasoning: $i, j \geq 0 \wedge i + j = 0$ ∗)
$\exists r' \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^0} r' \wedge s \xrightarrow{\tau^0} s' \wedge q = r'][s'$
$\wedge i + j = 0$
$\Rightarrow$ (∗ Definition τ^0 ∗)
$\exists r' \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^0} r' \wedge s \xrightarrow{\tau^0} s' \wedge q = r'][s'$
$\wedge r = r' \wedge s = s'$
$\Rightarrow$ (∗ Definition τ^0, $q = r][s$ ∗)
$r][s \xrightarrow{\tau^0} q$

Induction step: $n = m + 1$ and assume that the lemma holds for m.

$\exists r' \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^i} r' \wedge s \xrightarrow{\tau^j} s' \wedge q = r'][s'$
$\wedge n = i + j$
$\Rightarrow$ (∗ Definition $\rightarrow$ ∗)
$\exists r', r_1, \in Q_r, s' \in Q_s, i, j \geq 0 : r \xrightarrow{\tau^{i-1}} r_1 \xrightarrow{\tau} r' \wedge s \xrightarrow{\tau^j} s'$
$\wedge q = r'][s' \wedge n = i + j$
$\Rightarrow$ (∗ Induction, $m = n - 1 = i + j - 1$ ∗)
$\exists r', r_1 \in Q_r, s' \in Q_s, i, j \geq 0 : r_1 \xrightarrow{\tau} r' \wedge q = r'][s'$
$\wedge n = i + j \wedge r][s \xrightarrow{\tau^{i+j-1}} r_1][s'$

$$\begin{array}{ll}
\Rightarrow & (*\ \text{Lemma A.4.1 case 3}\ *) \\
& \exists r', r_1 \in Q_r, s' \in Q_s, i, j \geq 0 : q = r']\![s' \wedge n = i + j \\
& \wedge\, r]\![s \xrightarrow{\tau^{i+j-1}} r_1]\![s' \xrightarrow{\tau} r']\![s' \\
\Rightarrow & (*\ \text{Logical reasoning}\ *) \\
& r]\![s \xrightarrow{\tau^n} q
\end{array}$$

In case $i = 0$ the above splitting of τ^i does not work. In this case we split τ^j; the proof is symmetrical to the one above.

□

Lemma A.4.3 Let $r \in \mathbf{LTS}(I_r, U_r)$, $s \in \mathbf{LTS}(I_s, U_s)$.

1. $\forall a \in L_r \backslash L_s : r]\![s \overset{a}{\Longrightarrow} q \Leftrightarrow \exists r' \in Q_r, s' \in Q_s : r \overset{a}{\Longrightarrow} r' \wedge s \overset{\epsilon}{\Longrightarrow} s' \wedge q = r']\![s'$

2. $\forall a \in L_s \backslash L_r : r]\![s \overset{a}{\Longrightarrow} q \Leftrightarrow \exists r' \in Q_r, s' \in Q_s : r \overset{\epsilon}{\Longrightarrow} r' \wedge s \overset{a}{\Longrightarrow} s' \wedge q = r']\![s'$

3. $r]\![s \overset{\epsilon}{\Longrightarrow} q \Leftrightarrow \exists r' \in Q_r, s' \in Q_s : r \overset{\epsilon}{\Longrightarrow} r' \wedge s \overset{\epsilon}{\Longrightarrow} s' \wedge q = r']\![s'$

4. $\forall x \in U_r \cap I_s : r]\![s \overset{x}{\Longrightarrow} q \Leftrightarrow (\exists r' \in Q_r, s' \in Q_s : r \overset{x}{\Longrightarrow} r' \wedge s \overset{x}{\Longrightarrow} s' \wedge q = r']\![s') \vee (\exists r' \in Q_r, s' \in Q_s : r \overset{x}{\Longrightarrow} r' \wedge s \overset{\epsilon}{\Longrightarrow} s' \wedge s' \overset{\tau}{\not\longrightarrow} \wedge s' \overset{x}{\not\longrightarrow} \wedge q \in \{r']\![\chi(s)), r']\![\Omega(s), r']\![\Delta(s)\}$

5. $\forall x \in I_r \cap U_s : r]\![s \overset{x}{\Longrightarrow} q \Leftrightarrow (\exists r' \in Q_r, s' \in Q_s : r \overset{x}{\Longrightarrow} r' \wedge s \overset{x}{\Longrightarrow} s' \wedge q = r']\![s') \vee (\exists r' \in Q_r, s' \in Q_s : r \overset{\epsilon}{\Longrightarrow} r' \wedge r' \overset{\tau}{\not\longrightarrow} \wedge r' \overset{x}{\not\longrightarrow} \wedge s \overset{x}{\Longrightarrow} s' \wedge q \in \{\chi(r)]\![s', \Delta(r)]\![s', \Omega(r)]\![s'\}$

6. $r]\![s \overset{\delta}{\Longrightarrow} q \Leftrightarrow \exists r' \in Q_r, s' \in Q_s : r \overset{\delta}{\Longrightarrow} r' \wedge s \overset{\delta}{\Longrightarrow} s' \wedge q = r']\![s'$

□

Proof We begin with the proof of case 3, because we can use this result in the other proofs. Lemma A.4.2 proves a stronger case.
Proof of case 1.

Only if:

$$\begin{array}{ll}
& r]\![s \overset{a}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
& \exists q_1, q_2 \in Q_{r]\![s} : r]\![s \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{Lemma A.4.3 case 3}\ *) \\
& \exists q_1, q_2 \in Q_{r]\![s} : q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Longrightarrow} q \\
& \wedge\, \exists r_1 \in Q_r, s_1 \in Q_s : r \overset{\epsilon}{\Longrightarrow} r_1 \wedge s \overset{\epsilon}{\Longrightarrow} s_1 \wedge q_1 = r_1]\![s_1
\end{array}$$

$\Rightarrow$ (* Lemma A.4.1 case 1 *)
$\exists q_2 \in Q_{r][s} : q_2 \stackrel{\epsilon}{\Longrightarrow} q \wedge \exists r_1, r_2 \in Q_r, s_1 \in Q_s : r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{a}{\longrightarrow} r_2$
$\wedge\, s \stackrel{\epsilon}{\Longrightarrow} s_1 \wedge q_2 = r_2][s_1$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists r_1, r_2, r' \in Q_r, s_1, s' \in Q_s : r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{a}{\longrightarrow} r_2 \stackrel{\epsilon}{\Longrightarrow} r'$
$\wedge\, s \stackrel{\epsilon}{\Longrightarrow} s_1 \stackrel{\epsilon}{\Longrightarrow} s' \wedge q = r'][s'$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists r' \in Q_r, s' \in Q_s : r \stackrel{a}{\Longrightarrow} r' \wedge s \stackrel{\epsilon}{\Longrightarrow} s' \wedge q = r'][s'$

If:

$r \stackrel{a}{\Longrightarrow} r' \wedge s \stackrel{\epsilon}{\Longrightarrow} s'$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists r_1, r_2 \in Q_r : r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{a}{\longrightarrow} r_2 \stackrel{\epsilon}{\Longrightarrow} r' \wedge s \stackrel{\epsilon}{\Longrightarrow} s'$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists r_1, r_2 \in Q_r : r_1 \stackrel{a}{\longrightarrow} r_2 \stackrel{\epsilon}{\Longrightarrow} r' \wedge r][s \stackrel{\epsilon}{\Longrightarrow} r_1][s'$
$\Rightarrow$ (* Lemma A.4.1 case 1 *)
$\exists r_1, r_2 \in Q_r : r_2 \stackrel{\epsilon}{\Longrightarrow} r' \wedge r][s \stackrel{\epsilon}{\Longrightarrow} r_1][s' \stackrel{a}{\longrightarrow} r_2][s'$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists r_1, r_2 \in Q_r : r][s \stackrel{\epsilon}{\Longrightarrow} r_1][s' \stackrel{a}{\longrightarrow} r_2][s' \stackrel{\epsilon}{\Longrightarrow} r'][s'$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$r][s \stackrel{a}{\Longrightarrow} r'][s'$

Case 2: This case is symmetrical to case 1.
Case 4:

Only if:

$r][s \stackrel{x}{\Longrightarrow} q$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists q_1, q_2 \in Q_{r][s} : r][s \stackrel{\epsilon}{\Longrightarrow} q_1 \stackrel{x}{\longrightarrow} q_2 \stackrel{\epsilon}{\Longrightarrow} q$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists q_1, q_2 \in Q_{r][s} : q_1 \stackrel{x}{\longrightarrow} q_2 \stackrel{\epsilon}{\Longrightarrow} q$
$\wedge\, \exists r_1 \in Q_r : r \stackrel{\epsilon}{\Longrightarrow} r_1 \wedge \exists s_1 \in Q_s : s \stackrel{\epsilon}{\Longrightarrow} s_1 \wedge q_1 = r_1][s_1$
$\Rightarrow$ (* Lemma A.4.1, case 4 *)
$\exists q_2 \in Q_{r][s} : q_2 \stackrel{\epsilon}{\Longrightarrow} q \wedge (\exists r_1, r_2 \in Q_r, s_1, s_2 \in Q_s : r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{x}{\longrightarrow} r_2$
$\wedge\, s \stackrel{\epsilon}{\Longrightarrow} s_1 \stackrel{x}{\longrightarrow} s_2 \wedge q_2 = r_2][s_2) \vee (\exists r_1, r_2 \in Q_r, s_1 \in Q_s :$
$r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{x}{\longrightarrow} r_2 \wedge s \stackrel{\epsilon}{\Longrightarrow} s_1 \wedge s_1 \stackrel{x}{\not\longrightarrow} \wedge\, s_1 \stackrel{\tau}{\not\longrightarrow} \wedge\, q_2 = r_2][\chi(s))$
$\Rightarrow$ (* Lemma A.4.3 case 3, $\Delta(s)$, $\Omega(s)$ are reachable
via τ-steps (definition $\chi(s)$) *)
$(\exists r_1, r_2, r' \in Q_r, s_1, s_2, s' \in Q_s : r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{x}{\longrightarrow} r_2 \stackrel{\epsilon}{\Longrightarrow} r'$
$\wedge\, s \stackrel{\epsilon}{\Longrightarrow} s_1 \stackrel{x}{\longrightarrow} s_2 \stackrel{\epsilon}{\Longrightarrow} s' \wedge q = r'][s') \vee (\exists r_1, r_2 \in Q_r, s_1 \in Q_s :$
$r \stackrel{\epsilon}{\Longrightarrow} r_1 \stackrel{x}{\longrightarrow} r_2 \stackrel{\epsilon}{\Longrightarrow} r' \wedge s \stackrel{\epsilon}{\Longrightarrow} s_1 \wedge s_1 \stackrel{x}{\not\longrightarrow} \wedge\, s_1 \stackrel{\tau}{\not\longrightarrow}$
$\wedge\, q \in \{r'][\chi(s)), r'][\Delta(s), r'][\Omega(s)\}$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$(\exists r' \in Q_r, s' \in Q_s : r \overset{x}{\Longrightarrow} r' \wedge s \overset{x}{\Longrightarrow} s' \wedge q = r'][s')$
$\vee (\exists r' \in Q_r, s' \in Q_s : r \overset{x}{\Longrightarrow} r' \wedge s \overset{\epsilon}{\Longrightarrow} s' \wedge s' \overset{x}{\not\longrightarrow} \wedge s' \overset{\tau}{\not\longrightarrow}$
$\wedge q \in \{r'][\chi(s)), r'][\Delta(s), r'][\Omega(s)\}$

If: We split this proof in two parts, one for $s \overset{x}{\Longrightarrow} s'$ and one for $\exists s' \in Q_s : s \overset{\epsilon}{\Longrightarrow} s' \wedge s' \overset{\tau}{\not\longrightarrow} \wedge s' \overset{x}{\not\longrightarrow}$

$r \overset{x}{\Longrightarrow} r' \wedge s \overset{x}{\Longrightarrow} s'$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists r_1, r_2 \in Q_r : r \overset{\epsilon}{\Longrightarrow} r_1 \overset{x}{\longrightarrow} r_2 \overset{\epsilon}{\Longrightarrow} r$
$\wedge \exists s_1, s_2 \in Q_s : s \overset{\epsilon}{\Longrightarrow} s_1 \overset{x}{\longrightarrow} s_2 \overset{\epsilon}{\Longrightarrow} s'$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists r_1, r_2 \in Q_r : r_1 \overset{x}{\longrightarrow} r_2 \overset{\epsilon}{\Longrightarrow} r \wedge \exists s_1, s_2 \in Q_s : s_1 \overset{x}{\longrightarrow} s_2 \overset{\epsilon}{\Longrightarrow} s'$
$\wedge r][s \overset{\epsilon}{\Longrightarrow} r_1][s_1$
$\Rightarrow$ (* Lemma A.4.1 case 4 *)
$\exists r_1, r_2 \in Q_r : r_2 \overset{\epsilon}{\Longrightarrow} r \wedge \exists s_1, s_2 \in Q_s : s_2 \overset{\epsilon}{\Longrightarrow} s'$
$\wedge r][s \overset{\epsilon}{\Longrightarrow} r_1][s_1 \overset{x}{\longrightarrow} r_2][s_2$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists r_1, r_2 \in Q_r, s_1, s_2 \in Q_s : r][s \overset{\epsilon}{\Longrightarrow} r_1][s_1 \overset{x}{\longrightarrow} r_2][s_2 \overset{\epsilon}{\Longrightarrow} r'][s'$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$r][s \overset{x}{\Longrightarrow} r'][s'$

We continue with the case for $\exists s' : s \overset{\epsilon}{\Longrightarrow} s' \wedge s' \overset{\tau}{\not\longrightarrow} \wedge s' \overset{x}{\not\longrightarrow}$.

$r \overset{x}{\Longrightarrow} r' \wedge \exists s_1 \in Q_s : s \overset{\epsilon}{\Longrightarrow} s_1 \wedge s_1 \overset{\tau}{\not\longrightarrow} \wedge s_1 \overset{x}{\not\longrightarrow}$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists r_1, r_2 \in Q_r : r \overset{\epsilon}{\Longrightarrow} r_1 \overset{x}{\longrightarrow} r_2 \overset{\epsilon}{\Longrightarrow} r'$
$\wedge \exists s_1 \in Q_s : s \overset{\epsilon}{\Longrightarrow} s_1 \wedge s_1 \overset{x}{\not\longrightarrow} \wedge s_1 \overset{\tau}{\not\longrightarrow}$
$\Rightarrow$ (* Lemma A.4.3 case 3 *)
$\exists r_1, r_2 \in Q_r : r_1 \overset{x}{\longrightarrow} r_2 \overset{\epsilon}{\Longrightarrow} r'$
$\wedge \exists s_1 \in Q_s : s_1 \overset{x}{\not\longrightarrow} \wedge s_1 \overset{\tau}{\not\longrightarrow} \wedge r][s \overset{\epsilon}{\Longrightarrow} r_1][s_1$
$\Rightarrow$ (* Lemma A.4.1 case 4 *)
$\exists r_1, r_2 \in Q_r : r_2 \overset{\epsilon}{\Longrightarrow} r' \wedge r][s \overset{\epsilon}{\Longrightarrow} r_1][s_1 \overset{x}{\longrightarrow} r_2][\chi(s)$
$\Rightarrow$ (* Lemma A.4.3 case 3, $\Delta(s), \Omega(s)$ are reachable via τ-steps (definition $\chi(s)$) *)
$\exists r_1, r_2 \in Q_r : r][s \overset{\epsilon}{\Longrightarrow} r_1][s_1 \overset{x}{\longrightarrow} r_2][\chi(s) \overset{\epsilon}{\Longrightarrow} q$
$\wedge q \in \{r'][\chi(s), r'][\Delta(s), r'][\Omega(s)\}$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$r][s \overset{x}{\Longrightarrow} q \wedge q \in \{r'][\chi(s), r'][\Delta(s), r'][\Omega(s)\}$

Case 5: This case is symmetrical to case 4.

Case 6:

Only if:

$$
\begin{array}{ll}
 & r][s \overset{\delta}{\Longrightarrow} q \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow \text{ and definition } \delta\ *) \\
 & \exists q \in Q_{r][s} : r][s \overset{\epsilon}{\Longrightarrow} q \xrightarrow{\delta} q \\
\Rightarrow & (*\ \text{Lemma A.4.3 case 3}\ *) \\
 & \exists q \in Q_{r][s} : q \xrightarrow{\delta} q \\
 & \wedge \exists r_1 \in Q_r : r \overset{\epsilon}{\Longrightarrow} r_1 \wedge \exists s_1 \in Q_s : s \overset{\epsilon}{\Longrightarrow} s_1 \wedge q = r_1][s_1 \\
\Rightarrow & (*\ \text{Lemma A.4.1, case 6}\ *) \\
 & \exists r_1 \in Q_r : r \overset{\epsilon}{\Longrightarrow} r_1 \xrightarrow{\delta} r_1 \\
 & \wedge \exists s_1 \in Q_s : s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{\delta} s_1 \wedge q = r_1][s_1 \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists r' \in Q_r, s' \in Q_s : r \overset{\delta}{\Longrightarrow} r' \wedge s \overset{\delta}{\Longrightarrow} s' \wedge q = r'][s'
\end{array}
$$

If:

$$
\begin{array}{ll}
 & r \overset{\delta}{\Longrightarrow} r' \wedge s \overset{\delta}{\Longrightarrow} s' \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow \text{ and definition } \delta\ *) \\
 & r \overset{\epsilon}{\Longrightarrow} r' \xrightarrow{\delta} r' \wedge s \overset{\epsilon}{\Longrightarrow} s' \xrightarrow{\delta} s' \\
\Rightarrow & (*\ \text{Lemma A.4.3 case 3}\ *) \\
 & r' \xrightarrow{\delta} r' \wedge s' \xrightarrow{\delta} s' \wedge r][s \overset{\epsilon}{\Longrightarrow} r'][s' \\
\Rightarrow & (*\ \text{Lemma A.4.1 case 6}\ *) \\
 & r][s \overset{\epsilon}{\Longrightarrow} r'][s' \xrightarrow{\delta} r'][s' \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & r][s \overset{\delta}{\Longrightarrow} r'][s'
\end{array}
$$

□

Lemma A.4.4 Let $s \in \mathbf{LTS}$.

1. $\forall \lambda \in L : \chi(s) \overset{\lambda}{\Longrightarrow} \chi(s)$
2. $\chi(s) \overset{\delta}{\Longrightarrow} \Delta(s)$
3. $\forall a \in I : \Delta(s) \xrightarrow{a} \chi(s)$

□

Proof All cases follow from the definition of the chaotic transition system. From χ we arrive at Ω via an internal step and Ω can perform all actions in L. Likewise for the second case. From χ we arrive via an internal step in Δ, this state can only perform input actions (case 3) and hence is quiescent. □

The following lemma's help with properties about the projection of traces onto a label set.

Lemma A.4.5 *A and B are label-sets with $A \subseteq B$. Let $\sigma_1 \in B^*$, $\sigma_2 \in A^*$*

$$(\sigma_1 \cdot \sigma_2)\restriction A = (\sigma_1 \restriction A) \cdot \sigma_2$$

□

Proof From Definition 2.3.5 (projection) it is immediately clear that $\sigma_2 \restriction A = \sigma_2$, because $\sigma_2 \in A^*$. □

Lemma A.4.6 *A and B are label-sets with $A \cap B = \emptyset$. Let $\sigma_1 \in A^*, \sigma_2 \in B^*$*

$$(\sigma_1 \cdot \sigma_2)\restriction A = \sigma_1 \restriction A$$

□

Proof From Definition 2.3.5 (projection) it is immediately clear that $\sigma_2 \restriction A = \epsilon$. $\sigma_2 = b_1 \cdots b_n$ for some $n \geq 0$ and $\forall 0 \leq i \leq n : b_i \notin A$. □

Lemma A.4.7 *A is a label-set. Let $\sigma_1, \sigma_2 \in A^*$*

$$(\sigma_1 \cdot \sigma_2)\restriction A = (\sigma_1 \restriction A) \cdot (\sigma_2 \restriction A)$$

□

Proof This follows directly from the definition of projection for traces (Definition 2.3.5). Suppose that $\sigma_1 = \lambda_1 \cdots \lambda_n$, for some $n \geq 0$ with $\lambda_i \in A$ for $0 \leq i \leq n$. Likewise suppose that $\sigma_2 = \mu_1 \cdots \mu_m$, for some $m \geq 0$ with $\mu_i \in A$ for $0 \leq i \leq m$.

$$\begin{array}{ll} & (\sigma_1 \cdot \sigma_2)\restriction A \\ = & (*\ \text{Premise } \sigma_1 \text{ and } \sigma_2\ *) \\ & (\lambda_1 \cdots \lambda_n \cdot \mu_1 \cdots \mu_m)\restriction A \\ = & (*\ \text{Definition 2.3.5}\ *) \\ & \lambda_1 \restriction A \cdots \lambda_n \restriction A \cdot \mu_1 \restriction A \cdots \mu_m \restriction A \\ = & (*\ \text{Definition 2.3.5}\ *) \\ & (\lambda_1 \cdots \lambda_n)\restriction A \cdot (\mu_1 \cdots \mu_m)\restriction A \\ = & (*\ \text{Premise } \sigma_1 \text{ and } \sigma_2\ *) \\ & \sigma_1 \restriction A \cdot \sigma_2 \restriction A \end{array}$$

□

Proposition A.4.8 *Let $r \in \mathbf{LTS}(I_r, U_r), s \in \mathbf{LTS}(I_s, U_s)$ with $\sigma \in (L_r \cup L_s \cup \{\delta\})^*$*

$$\exists s' \in Q_s, r' \in Q_r : r \xRightarrow{\sigma \restriction L_r^\delta} r' \wedge s \xRightarrow{\sigma \restriction L_s^\delta} s' \Rightarrow r][s \xRightarrow{\sigma} r'][s'$$

□

Proof Proof by induction on the length of σ

Basic step: $\sigma = \epsilon$. This case is proven in Lemma A.4.3 case 3.

Induction step: Let $\sigma = \sigma' \cdot \lambda$. We assume that the lemma holds for σ'. Based on Lemma A.4.3 we identify the following cases: $\lambda \in L_r \backslash L_s$, $\lambda \in L_s \backslash L_r$, $\lambda \in L_r \cap L_s$, $\lambda = \delta$. We start with $\lambda \in L_r \backslash L_s$.

$r \xRightarrow{(\sigma' \cdot \lambda) \lceil L_r^\delta} r' \wedge s \xRightarrow{(\sigma' \cdot \lambda) \lceil L_s^\delta} s'$

$\Rightarrow$ (* Lemma A.4.5 and Lemma A.4.6 (projection) *)
$r \xRightarrow{\sigma' \lceil L_r^\delta \cdot \lambda} r' \wedge s \xRightarrow{\sigma' \lceil L_s^\delta} s'$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists r_1 \in Q_r : r \xRightarrow{\sigma' \lceil L_r^\delta} r_1 \xRightarrow{\lambda} r' \wedge s \xRightarrow{\sigma' \lceil L_s^\delta} s'$

$\Rightarrow$ (* Induction *)
$\exists r_1 \in Q_r : r_1 \xRightarrow{\lambda} r' \wedge r][s \xRightarrow{\sigma'} r_1][s'$

$\Rightarrow$ (* Lemma A.4.3 case 1 *)
$\exists r_1 \in Q_r : r][s \xRightarrow{\sigma'} r_1][s' \xRightarrow{\lambda} r'][s'$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$r][s \xRightarrow{\sigma' \cdot \lambda} r'][s'$

The case $\lambda \in L_s \backslash L_r$ is symmetrical to the previous one, using Lemma A.4.3 case 2. We continue with the case $\lambda \in L_r \cap L_s$.

$r \xRightarrow{(\sigma' \cdot \lambda) \lceil L_r^\delta} r' \wedge s \xRightarrow{(\sigma' \cdot \lambda) \lceil L_s^\delta} s'$

$\Rightarrow$ (* Lemma A.4.5 (projection) *)
$s \xRightarrow{\sigma' \lceil L_s^\delta \cdot \lambda} r' \wedge s \xRightarrow{\sigma' \lceil L_s^\delta \cdot \lambda} s'$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists r_1 \in Q_r, s_1 \in Q_s : r \xRightarrow{\sigma' \lceil L_r^\delta} r_1 \xRightarrow{\lambda} r' \wedge s \xRightarrow{\sigma' \lceil L_s^\delta} s_1 \xRightarrow{\lambda} s'$

$\Rightarrow$ (* Induction *)
$\exists r_1 \in Q_r, s_1 \in Q_s : r_1 \xRightarrow{\lambda} r' \wedge s_1 \xRightarrow{\lambda} s' \wedge r][s \xRightarrow{\sigma'} r_1][s_1$

$\Rightarrow$ (* Lemma A.4.3 cases 4 and 5 *)
$\exists r_1 \in Q_r, s_1 \in Q_s : r][s \xRightarrow{\sigma'} r_1][s_1 \xRightarrow{\lambda} r'][s'$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$r][s \xRightarrow{\sigma' \cdot \lambda} r'][s'$

We continue with the case $\lambda = \delta$: The proof of this case is identical to the previous one, where the fourth proof step (Lemma A.4.3 cases 4 and 5) is replaced by (Lemma A.4.3 case 6).

□

Lemma A.4.9

$$\forall r \in \mathbf{LTS}(I, U) : \mathit{Straces}(r) \subseteq \mathit{Straces}(\chi(r))$$

□

Proof Proof by reductio ad absurdum. Suppose that this lemma does not hold. This means that there is a state in Q_r that can perform an action $\lambda \in L_\delta$ that the state that we reached in $Q_{\chi(s)}$ cannot perform. We check the actions possible in the three states of $Q_{\chi(s)}$:

- $\chi(r)$. Lemma A.4.4 shows that χ can perform all actions in L_δ.
- $\Omega(r)$. Lemma A.4.4 shows that Ω can perform all actions in L. This means that the only candidate action that Ω cannot do is δ. However we get to Ω via an internal action from χ (which means that χ is also reachable) and χ can do all actions in L_δ.
- $\Delta(r)$. Lemma A.4.4 shows that Δ can perform all actions in I_δ. We get to Δ via an internal action from χ (which means that χ is also reachable), this means that we can do all actions in L_δ. Δ is reached via a δ transition from χ. What if r can do an output action at this point? This case leads to a contradiction, because by definition output actions are not possible after δ.

□

Lemma A.4.10 Let $r, s \in \mathbf{LTS}$, $\sigma \in (L_r \cup L_s \cup \{\delta\})^*$.

$$r \xRightarrow{\sigma \restriction L_r^\delta} \wedge\, \exists t \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma \restriction L_s^\delta} \Rightarrow r][\chi(s) \xRightarrow{\sigma}$$

□

Proof

$\Rightarrow$ (∗ Lemma A.4.9 ∗)
$r \xRightarrow{\sigma \restriction L_r^\delta} \wedge\, \chi(s) \xRightarrow{\sigma \restriction L_s^\delta}$
$\Rightarrow$ (∗ Proposition A.4.8 ∗)
$r][\chi(s) \xRightarrow{\sigma}$

□

The following lemma treats the case when a suspension trace from a composition of two systems is not a suspension trace of one of the components (the projection onto its label set). In this case the culprit is an input action that the respective component cannot perform.

Lemma A.4.11 Let $r, s \in \mathbf{LTS}, \sigma{\cdot}\lambda \in \mathit{Straces}(r][s)$, $\sigma \restriction L_s^\delta \in \mathit{Straces}(s)$

$$\sigma \restriction L_r^\delta \in \mathit{Straces}(r) \wedge (\sigma{\cdot}\lambda) \restriction L_r^\delta \notin \mathit{Straces}(r) \Rightarrow \lambda \in I_r \cap U_s$$

□

Proof The trace $\sigma{\cdot}\lambda$ is a trace of the composition of r and s. Somehow the projected trace is not a trace of the component r. This situation is only possible when an unspecified input action causes the composition to make a transition to a chaotic state.

$$\begin{array}{ll} & \forall r' \in (r \textbf{ after } \sigma{\restriction}L_r^\delta) : r' \overset{\lambda}{\not\Longrightarrow} \wedge \exists s' \in Q_s : s \xRightarrow{\sigma\restriction L_s^\delta} s' \\ & \wedge\, \sigma{\restriction}L_r^\delta \in \mathit{Straces}(r) \\ \Rightarrow & (*\ \text{Proposition A.4.8}\ *) \\ & \forall r' \in (r \textbf{ after } \sigma{\restriction}L_r^\delta) : r' \overset{\lambda}{\not\Longrightarrow} \wedge r][s \xRightarrow{\sigma} r'][s' \end{array}$$

We use Lemma A.4.3 to determine the nature of λ.

1. Case 1 is not applicable, because it assumes that r can perform the λ transition.

2. Case 2 leads to a contradiction. When $\lambda \in L_s \backslash L_r$, this means that λ is not a label of r, hence $\sigma{\restriction}L_r^\delta = \sigma{\cdot}\lambda{\restriction}L_r^\delta$ (Lemma A.4.5). However $\sigma{\restriction}L_r^\delta \in \mathit{Straces}(r)$, but $\sigma{\cdot}\lambda{\restriction}L_r^\delta \notin \mathit{Straces}(r)$.

3. Case 3 is not applicable, because λ is a label.

4. Case 4 is not applicable, because it assumes that r should be able to perform λ

5. Case 5 is applicable. It shows that the composition can perform λ even when r cannot perform λ. This implies that that $\lambda \in I_r \cap U_s$

6. Case 6 is not applicable, because it assumes that r can perform λ.

□

Proposition A.4.12 Let $r, s \in \textbf{IOTS}$

$$r][s \xRightarrow{\sigma} q \Rightarrow \exists r' \in Q_r, s' \in Q_s : r \xRightarrow{\sigma\restriction L_r^\delta} r' \wedge s \xRightarrow{\sigma\restriction L_s^\delta} s' \wedge q = r'][s'$$

□

Proof The parallel operator][is equivalent to the standard parallel operator for input enabled transition systems like the IOTSs. See Proposition A.1.2 for the proof. □

Lemma A.4.13 Let $r \in \textbf{LTS}(I_r, U_r), s \in \textbf{LTS}(I_s, U_s), \sigma \in (L_r \cup L_s \cup \{\delta\})^*, a \in I_s \cap U_r$.

$$r \xRightarrow{\sigma\cdot a\restriction L_r^\delta} r' \wedge s \xRightarrow{\sigma\restriction L_s^\delta} \wedge s \overset{\sigma\cdot a\restriction L_s^\delta}{\not\Longrightarrow} \Rightarrow r'][\chi(s) \in r][s \textbf{ after } \sigma{\cdot}a$$

□

Proof

$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists r_1 \in Q_r : r \xRightarrow{\sigma \restriction L_r^\delta} r_1 \xRightarrow{a \restriction L_r^\delta} r' \wedge \exists s_1 \in Q_s : s \xRightarrow{\sigma \restriction L_s^\delta} s_1 \not\xRightarrow{a \restriction L_s^\delta}$

$\Rightarrow$ (∗ Definition $\Rightarrow$, not that the transition systems are convergent and exhibit only a finite number of τ steps ∗)
$\exists r_1 \in Q_r : r \xRightarrow{\sigma \restriction L_r^\delta} r_1 \xRightarrow{a \restriction L_r^\delta} r' \wedge \exists s_1, s_2 \in Q_s : s \xRightarrow{\sigma \restriction L_s^\delta} s_1 \xRightarrow{\epsilon} s_2 \not\xrightarrow{a \restriction L_s^\delta, \tau}$

$\Rightarrow$ (∗ Definition $\Rightarrow$ ∗)
$\exists r_1 \in Q_r : r \xRightarrow{\sigma \restriction L_r^\delta} r_1 \xRightarrow{a \restriction L_r^\delta} r' \wedge \exists s_1 \in Q_s : s \xRightarrow{\sigma \restriction L_s^\delta} s_1 \not\xrightarrow{a \restriction L_s^\delta, \tau}$

$\Rightarrow$ (∗ Proposition A.4.8 (component composition) ∗)
$\exists r_1 \in Q_r : r_1 \xRightarrow{a \restriction L_r^\delta} r' \wedge \exists s_1 \in Q_s : s_1 \not\xrightarrow{a \restriction L_s^\delta, \tau} \wedge r][s \xRightarrow{\sigma} r_1][s_1$

$\Rightarrow$ (∗ Lemma A.4.3 case 4 ∗)
$\exists r_1 \in Q_r, s_1 \in Q_s, q \in \{r'][\chi(s), r'][\Delta(s), r'][\Omega(s)\} :$
$r][s \xRightarrow{\sigma} r_1][s_1 \xRightarrow{a} q$

$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists q \in \{r'][\chi(s), r'][\Delta(s), r'][\Omega(s)\} : r][s \xRightarrow{\sigma \cdot a} q$

$\Rightarrow$ (∗ Definition $\chi(s)$: $\Delta(s)$, $\Omega(s)$ are only reachable via $\chi(s)$, together with definition **after** ∗)
$r'][\chi(s) \in r][s$ **after** $\sigma \cdot a$

□

Lemma A.4.14 Let $\sigma \in \mathit{Straces}(r][s)$.

$$\begin{aligned}
&\sigma \restriction L_r^\delta \in \mathit{Straces}(r) \wedge \sigma \restriction L_s^\delta \notin \mathit{Straces}(s)\\
&\quad \Rightarrow \exists r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2\\
&\quad \wedge r \xRightarrow{\sigma_1 \restriction L_r^\delta} r' \xRightarrow{\sigma_2 \restriction L_r^\delta} \wedge r'][\chi(s) \in r][s \textbf{ after } \sigma_1
\end{aligned}$$

□

Proof We split σ into $\sigma_1 \cdot a \cdot \sigma_2$ such that $\sigma_1 \restriction L_s^\delta$ is the longest prefix that is still a suspension trace of s. This means that action a is the first action that s cannot perform anymore, as a result causing the s component to become chaotic.

$\Rightarrow$ (∗ We split σ into $\sigma_1 \cdot a \cdot \sigma_2$ ∗)
$\exists \sigma_1, \sigma_2, a : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \wedge \sigma_1 \restriction L_s^\delta \in \mathit{Straces}(s) \wedge \sigma_1 \cdot a \restriction L_s^\delta \notin \mathit{Straces}(s)$
$\wedge r \xRightarrow{\sigma_1 \cdot a \cdot \sigma_2 \restriction L_r^\delta}$

$\Rightarrow$ (∗ Lemma A.4.11 (input actions trigger chaos) ∗)
$\exists \sigma_1, \sigma_2, a \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \wedge \sigma_1 \restriction L_s^\delta \in \mathit{Straces}(s)$
$\wedge \sigma_1 \cdot a \restriction L_s^\delta \notin \mathit{Straces}(s) \wedge r \xRightarrow{\sigma_1 \cdot a \cdot \sigma_2 \restriction L_r^\delta}$

$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists \sigma_1, \sigma_2, a \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \wedge \sigma_1 \restriction L_s^\delta \in \mathit{Straces}(s)$
$\wedge \sigma_1 \cdot a \restriction L_s^\delta \notin \mathit{Straces}(s) \wedge \exists r' \in Q_r : r \xRightarrow{\sigma_1 \cdot a \restriction L_r^\delta} r' \xRightarrow{\sigma_2 \restriction L_r^\delta}$

$\Rightarrow$ (∗ Lemma A.4.13 ∗)
$\exists \sigma_1, \sigma_2, a \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \wedge \exists r' \in Q_r : r \xRightarrow{\sigma_1 \cdot a \restriction L_r^\delta} r' \xRightarrow{\sigma_2 \restriction L_r^\delta}$
$\wedge r'][\chi(s) \in r][s$ **after** $\sigma_1 \cdot a$

$\Rightarrow$ (* Rename $\sigma_1{\cdot}a$ to σ_1 *)
$\exists r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1{\cdot}\sigma_2 \wedge r \xRightarrow{\sigma_1 \lceil L_r^\delta} r' \xRightarrow{\sigma_2 \lceil L_r^\delta}$
$\wedge\, r'\rfloor[\chi(s) \in r\rfloor[s$ **after** σ_1

□

Lemma A.4.15 Let $r \in \mathbf{LTS}(I_r, U_r)$, $s \in \mathbf{LTS}(I_s, U_s)$, $\sigma \in \mathit{Straces}(r\rfloor[s)$, $\sigma\lceil L_r^\delta \notin \mathit{Straces}(r)$, $\sigma\lceil L_s^\delta \notin \mathit{Straces}(s)$, $x \in U_r \cup U_s \cup \{\delta\}$.

$$\exists t \in \mathbf{LTS}(I_r, U_r), v \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_r^\delta} \wedge\, v \xRightarrow{\sigma\cdot x\lceil L_s^\delta} \Rightarrow r\rfloor[s \xRightarrow{\sigma\cdot x}$$

□

Proof This proof is analogous to the proof of Lemma A.4.14. According to Lemma A.4.11 a component becomes chaotic after an unspecified input action. In this case we have two components that become chaotic, because $\sigma\lceil L_r^\delta \notin \mathit{Straces}(r)$ and $\sigma\lceil L_s^\delta \notin \mathit{Straces}(s)$. Because a component becomes chaotic after an unspecified input action, and because $I_r \cap I_s = \emptyset$ the components cannot become chaotic after the same trace. In the proof we treat the case that component r is the first to become chaotic (the proof is symmetrical for the case that s is the first to become chaotic). We split σ into $\sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b{\cdot}\sigma_3$ such that $\sigma_1\lceil L_r^\delta$ is the longest prefix that is still a suspension trace of r. In other words, $\sigma_1{\cdot}a\lceil L_r^\delta$ is not a suspension trace of r anymore. Furthermore $\sigma_1{\cdot}a{\cdot}\sigma_2$ is the longest prefix of σ that is still a suspension trace of s. This means that $\sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b\lceil L_s^\delta$ is not a suspension trace of s anymore.

$\Rightarrow$ (* We split σ *)
$\exists t \in \mathbf{LTS}(I_r, U_r), v \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_r^\delta} \wedge\, v \xRightarrow{\sigma\cdot x\lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a, b : \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b{\cdot}\sigma_3 \wedge \sigma_1\lceil L_r^\delta \in \mathit{Straces}(r)$
$\wedge\, \sigma_1{\cdot}a\lceil L_r^\delta \notin \mathit{Straces}(r) \wedge \sigma_1{\cdot}a{\cdot}\sigma_2\lceil L_s^\delta \in \mathit{Straces}(s)$
$\wedge\, \sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b \notin \mathit{Straces}(s)$

$\Rightarrow$ (* Lemma A.4.11 (input actions trigger chaos) *)
$\exists t \in \mathbf{LTS}(I_r, U_r), v \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_r^\delta} \wedge\, v \xRightarrow{\sigma\cdot x\lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b{\cdot}\sigma_3$
$\wedge\, \sigma_1\lceil L_r^\delta \in \mathit{Straces}(r) \wedge \sigma_1{\cdot}a\lceil L_r^\delta \notin \mathit{Straces}(r)$
$\wedge\, \sigma_1{\cdot}a{\cdot}\sigma_2\lceil L_s^\delta \in \mathit{Straces}(s) \wedge \sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b \notin \mathit{Straces}(s)$

$\Rightarrow$ (* Definition *Straces* *)
$\exists t \in \mathbf{LTS}(I_r, U_r), v \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_r^\delta} \wedge\, v \xRightarrow{\sigma\cdot x\lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b{\cdot}\sigma_3$
$\wedge\, \exists r' \in Q_r : r \xRightarrow{\sigma_1\lceil L_r^\delta} r' \wedge r \not\xRightarrow{\sigma_1\cdot a\lceil L_r^\delta}$
$\wedge\, \exists s' \in Q_s : s \xRightarrow{\sigma_1\cdot a\lceil L_s^\delta} s' \xRightarrow{\sigma_2\lceil L_s^\delta} \wedge\, s' \not\xRightarrow{\sigma_2\cdot b\lceil L_s^\delta}$

$\Rightarrow$ (* Lemma A.4.13 *)
$\exists t \in \mathbf{LTS}(I_r, U_r), v \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_r^\delta} \wedge\, v \xRightarrow{\sigma\cdot x\lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2{\cdot}b{\cdot}\sigma_3$
$\wedge\, \exists s' \in Q_s : s' \xRightarrow{\sigma_2\lceil L_s^\delta} \wedge\, s' \not\xRightarrow{\sigma_2\cdot b\lceil L_s^\delta}$
$\wedge\, \chi(r)\rfloor[s' \in r\rfloor[s$ **after** $\sigma_1{\cdot}a$

$\Rightarrow$ (* Lemma A.4.9, note that in case s is the first to become chaotic we would use LTS v *)
$\exists v \in \mathbf{LTS}(I_s, U_s) : v \xRightarrow{\sigma \cdot x \lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\wedge\, \exists s' \in Q_s : s' \xRightarrow{a \cdot \sigma_2 \lceil L_s^\delta} \wedge\, s' \not\xRightarrow{\sigma_2 \cdot b \lceil L_s^\delta}$
$\wedge\, \chi(r)][s' \in r][s \,\mathbf{after}\, \sigma_1 \cdot a \wedge \chi(r) \xRightarrow{\sigma_2 \cdot b \cdot \sigma_3 \cdot x \lceil L_r^\delta}$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists v \in \mathbf{LTS}(I_s, U_s) : v \xRightarrow{\sigma \cdot x \lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\wedge\, \exists s' \in Q_s : s' \xRightarrow{\sigma_2 \lceil L_s^\delta} \wedge\, s' \not\xRightarrow{\sigma_2 \cdot b \lceil L_s^\delta}$
$\wedge\, \chi(r)][s' \in r][s \,\mathbf{after}\, \sigma_1 \cdot a$
$\wedge\, \exists q \in Q_{\chi(r)} : \chi(r) \xRightarrow{\sigma_2 \cdot b \lceil L_r^\delta} q \xRightarrow{\sigma_3 \cdot x \lceil L_r^\delta}$
$\Rightarrow$ (* Lemma A.4.13 *)
$\exists v \in \mathbf{LTS}(I_s, U_s) : v \xRightarrow{\sigma \cdot x \lceil L_s^\delta}$
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\wedge\, \exists s' \in Q_s : \chi(r)][s' \in r][s \,\mathbf{after}\, \sigma_1 \cdot a$
$\wedge\, \exists q \in Q_\chi(r) : \chi(r) \xRightarrow{\sigma_2 \cdot b \lceil L_r^\delta} q \xRightarrow{\sigma_3 \cdot x \lceil L_r^\delta}$
$\wedge\, q][\chi(s) \in \chi(r)][s' \,\mathbf{after}\, \sigma_2 \cdot b$
$\Rightarrow$ (* Lemma A.4.9 *)
$\wedge\, \exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\wedge\, \exists s' \in Q_s : \chi(r)][s' \in r][s \,\mathbf{after}\, \sigma_1 \cdot a$
$\wedge\, \exists q \in Q_{\chi(r)} : \chi(r) \xRightarrow{\sigma_2 \cdot b \lceil L_r^\delta} q \xRightarrow{\sigma_3 \cdot x \lceil L_r^\delta}$
$\wedge\, q][\chi(s) \in \chi(r)][s' \,\mathbf{after}\, \sigma_2 \cdot b$
$\wedge\, \chi(s) \xRightarrow{\sigma_3 \cdot x \lceil L_s^\delta}$
$\Rightarrow$ (* Definition **after** *)
$\exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\exists s' \in Q_s : r][s \xRightarrow{\sigma_1 \cdot a} \chi(r)][s'$
$\wedge\, \exists q \in Q_{\chi(r)} : \chi(r) \xRightarrow{\sigma_2 \cdot b \lceil L_r^\delta} q \xRightarrow{\sigma_3 \cdot x \lceil L_r^\delta}$
$\wedge\, \chi(r)][s' \xRightarrow{\sigma_2 \cdot b} q][\chi(s)$
$\wedge\, \chi(s) \xRightarrow{\sigma_3 \cdot x \lceil L_s^\delta}$
$\Rightarrow$ (* Proposition A.4.8 *)
$\exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\wedge\, \exists q \in Q_{\chi(r)}, s' \in Q_s : r][s \xRightarrow{\sigma_1 \cdot a} \chi(r)][s' \wedge \chi(r)][s' \xRightarrow{\sigma_2 \cdot b} q][\chi(s)$
$\wedge\, q][\chi(s) \xRightarrow{\sigma_3 \cdot x}$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists \sigma_1, \sigma_2, \sigma_3, a \in I_r \cap U_s, b \in I_s \cap U_r : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$
$\wedge\, r][s \xRightarrow{\sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3 \cdot x}$
$\Rightarrow$ (* Logical reasoning (σ) *)
$r][s \xRightarrow{\sigma \cdot x}$

Note that the splitting of the projected traces is allowed because of Lemma A.4.7. We did not refer to the lemma in the proof, because we think that it would deteriorate the readability of the already lengthy proof. □

Lemma A.4.16 Let $r \in \mathbf{LTS}(I_r, U_r)$, $s \in \mathbf{LTS}(I_s, U_s)$, $\sigma \in \mathit{Straces}(r][s)$, $\sigma\lceil L_s^\delta \notin \mathit{Straces}(s)$ and $x \in U_r^\delta$

$$r \xRightarrow{\sigma\cdot x\lceil L_r^\delta} \wedge \exists t \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_s^\delta} \Rightarrow \sigma\cdot x \in \mathit{Straces}(r][s)$$

$\square$

Proof The lemma describes the case where the s component has become chaotic in the composition. Because the non-chaotic part r can perform the output action x, this means that the composition can also do the output (the chaotic component s will perform the required input if there is one).

$\Rightarrow$ (∗ Lemma A.4.14 ∗)
$\exists t \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_s^\delta} \wedge \exists \sigma_1, \sigma_2 : \sigma = \sigma_1\cdot\sigma_2$
$\wedge \exists r_1 \in Q_r : r \xRightarrow{\sigma_1\lceil L_r^\delta} r_1 \xRightarrow{\sigma_2\lceil L_r^\delta\cdot x} \wedge r_1][\chi(s) \in r][s \textbf{ after } \sigma_1$
$\Rightarrow$ (∗ Definition **after** ∗)
$\exists t \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\cdot x\lceil L_s^\delta} \wedge \exists \sigma_1, \sigma_2 : \sigma = \sigma_1\cdot\sigma_2$
$\wedge \exists r_1 \in Q_r : r \xRightarrow{\sigma_1\lceil L_r^\delta} r_1 \xRightarrow{\sigma_2\lceil L_r^\delta\cdot x} \wedge r][s \xRightarrow{\sigma_1} r_1][\chi(s)$
$\Rightarrow$ (∗ Lemma A.4.10 ∗)
$\exists \sigma_1, \sigma_2 : \sigma = \sigma_1\cdot\sigma_2 \wedge \exists r_1 \in Q_r : r][s \xRightarrow{\sigma_1} r_1][\chi(s) \wedge r_1][\chi(s) \xRightarrow{\sigma_2\cdot x}$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists \sigma_1, \sigma_2 : \sigma = \sigma_1\cdot\sigma_2 \wedge r][s \xRightarrow{\sigma_1\cdot\sigma_2\cdot x}$
$\Rightarrow$ (∗ Logical reasoning: $\sigma = \sigma_1\cdot\sigma_2$ ∗)
$r][s \xRightarrow{\sigma\cdot x}$

$\square$

Lemma A.4.17 Let $r \in \mathbf{LTS}(I_r, U_r)$, $s \in \mathbf{LTS}(I_s, U_s)$, $\sigma \in \mathit{Straces}(r][s)$, $\sigma\lceil L_r^\delta \in \mathit{Straces}(r)$, $\sigma\lceil L_s^\delta \notin \mathit{Straces}(s)$, $x \in U_s$

$$\exists t \in \mathbf{LTS}(I_s, U_s) : t \xRightarrow{\sigma\lceil L_s^\delta\cdot x} \Rightarrow \sigma\cdot x \in \mathit{Straces}(r][s)$$

$\square$

Proof Component s becomes chaotic in the composition. The chaotic system for s can do all output actions, for example x mentioned in the lemma. As a result the composition can also do the output action. If there is no corresponding input action x from r the composition can do the output action. If there is a corresponding input action x from r, and r can perform the action, the composition can also perform the action. In case r cannot perform the action, the component becomes chaotic, but the composition can do the output action. Like in the previous cases we split σ into $\sigma_1\cdot\sigma_2$ where σ_1 is the shortest prefix of σ that is not a suspension trace of s anymore.

$\Rightarrow$ (∗ Lemma A.4.14 ∗)
$\exists t \in \mathbf{LTS}(I_s, U_s), x \in U_s : t \xRightarrow{\sigma \upharpoonright L_s^\delta \cdot x}$
$\wedge\, \exists r_1, r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2 \wedge r_1 \xRightarrow{\sigma_2 \upharpoonright L_r^\delta} r'$
$\wedge\, r_1][\chi(s) \in r][s \textbf{ after } \sigma_1$
$\Rightarrow$ (∗ Definition **after** ∗)
$\exists t \in \mathbf{LTS}(I_s, U_s), x \in U_s : t \xRightarrow{\sigma \upharpoonright L_s^\delta \cdot x}$
$\wedge\, \exists r_1, r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2 \wedge r_1 \xRightarrow{\sigma_2 \upharpoonright L_r^\delta} r'$
$\wedge\, r][s \xRightarrow{\sigma_1} r_1][\chi(s)$
$\Rightarrow$ (∗ Lemma A.4.9 ∗)
$\exists r_1, r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2 \wedge r_1 \xRightarrow{\sigma_2 \upharpoonright L_r^\delta} r'$
$\wedge\, r][s \xRightarrow{\sigma_1} r_1][\chi(s) \wedge \chi(s) \xRightarrow{\sigma_2 \upharpoonright L_s^\delta \cdot x}$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists r_1, r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2 \wedge r_1 \xRightarrow{\sigma_2 \upharpoonright L_r^\delta} r'$
$\wedge\, r][s \xRightarrow{\sigma_1} r_1][\chi(s) \wedge \exists q \in Q_{\chi(s)} : \chi(s) \xRightarrow{\sigma_2 \upharpoonright L_s^\delta} q \xRightarrow{x}$
$\Rightarrow$ (∗ Proposition A.4.8 (composition) ∗)
$\exists r_1, r' \in Q_r, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2$
$\wedge\, \exists q \in Q_{\chi(s)} : q \xRightarrow{x} \wedge\, r][s \xRightarrow{\sigma_1} r_1][\chi(s) \xRightarrow{\sigma_2} r'][q$

At this point we identify the cases $x \in U_s \backslash L_r$ and $x \in I_r \cap U_s$. We continue the proof with the former.

$\Rightarrow$ (∗ Lemma A.4.3 case 2 ∗)
$\exists r_1, r' \in Q_r, q \in Q_\chi, \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2$
$\wedge\, r][s \xRightarrow{\sigma_1} r_1][\chi(s) \xRightarrow{\sigma_2} r'][q \xRightarrow{x}$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists \sigma_1, \sigma_2 : \sigma = \sigma_1 \cdot \sigma_2 \wedge r][s \xRightarrow{\sigma_1 \cdot \sigma_2 \cdot x}$
$\Rightarrow$ (∗ Logical reasoning ∗)
$r][s \xRightarrow{\sigma \cdot x}$

The case for $x \in I_r \cap U_s$ goes analogous using Lemma A.4.3 case 4. Note that we make use of the fact that the transitions systems are convergent (see Lemma A.4.13 for a similar case).

□

Theorem 3.4.11 Let $s_1, s_2 \in \mathbf{LTS}$ and $i_1, i_2 \in \mathbf{IOTS}$ with $I_{s_1} = I_{i_1}, U_{s_1} = U_{i_1}, I_{s_2} = I_{i_2}, U_{s_2} = U_{i_2}$.

$$i_1 \textbf{ ioco } s_1 \wedge i_2 \textbf{ ioco } s_2 \Rightarrow i_1][i_2 \textbf{ ioco } s_1][s_2$$

□

Proof The crux of this proof is that the parallel composition of s_1 and s_2 has state tuples that may consist of chaotic states. For non-chaotic states we use the **ioco** premise on the components to deduce which output actions are allowed in the respective components. In case the state is chaotic, we know that all output actions of that component are possible. When we expand the **ioco** definition we have to prove the following:

$$\forall \sigma \in \mathit{Straces}(s_1][s_2) : \mathit{out}(i_1][i_2 \textbf{ after } \sigma) \subseteq \mathit{out}(s_1][s_2 \textbf{ after } \sigma)$$

We start by showing that both implementation components can perform the x action if it is in their label-set. The trace σ in the proof is a suspension trace of $s_1][s_2$.

$$
\begin{array}{ll}
 & x \in out(i_1][i_2 \textbf{ after } \sigma) \\
\Rightarrow & (*\ \text{Definition } out \text{ and } \textbf{after} \ *) \\
 & i_1][i_2 \xRightarrow{\sigma\cdot x} \\
\Rightarrow & (*\ \text{Proposition A.4.12} \ *) \\
 & i_1 \xRightarrow{\sigma\cdot x\lceil L^{\delta}_{i_1}} \wedge i_2 \xRightarrow{\sigma\cdot x\lceil L^{\delta}_{i_2}}
\end{array}
$$

In other words, both components can perform the action when it is in their label set.

The distinction between projections that are suspension traces and projections that are not is important. If a projection is not a suspension trace of its respective component, this means that the component becomes chaotic in the composition. We identify the following cases:

1. $\sigma\lceil L^{\delta}_{s_1} \in \mathit{Straces}(s_1)$ and $\sigma\lceil L^{\delta}_{s_2} \in \mathit{Straces}(s_2)$.

   Because $\sigma\lceil L^{\delta}_{i_1}$ is a suspension trace of s_1 ($\sigma\lceil L^{\delta}_{s_1} = \sigma\lceil L^{\delta}_{i_1}$), we can use our **ioco** premise to figure out the output actions of s_1 (likewise for s_2).

   In case $x \in U_{s_1}$ we obtain the following result.

$$
\begin{array}{ll}
 & \sigma\lceil L^{\delta}_{s_1} \in \mathit{Straces}(s_1) \wedge \sigma\lceil L^{\delta}_{s_2} \in \mathit{Straces}(s_2) \\
 & \wedge\, x \in out(i_1 \textbf{ after } \sigma\lceil L^{\delta}_{i_1}) \\
\Rightarrow & (*\ i_1 \textbf{ ioco } s_1,\ L_{i_1} = L_{s_1} \ *) \\
 & x \in out(s_1 \textbf{ after } \sigma\lceil L^{\delta}_{s_1}) \wedge \sigma\lceil L^{\delta}_{s_2} \in \mathit{Straces}(s_2) \\
\Rightarrow & (*\ \text{Definitions } out,\ \textbf{after} \text{ and } \mathit{Straces} \ *) \\
 & \exists s_1' \in Q_{s_1}, s_2' \in Q_{s_2} : s_1 \xRightarrow{\sigma\lceil L^{\delta}_{s_1}} s_1' \xRightarrow{x} \wedge s_2 \xRightarrow{\sigma\lceil L^{\delta}_{s_2}} s_2' \\
\Rightarrow & (*\ \text{Proposition A.4.8} \ *) \\
 & \exists s_1' \in Q_{s_1}, s_2' \in Q_{s_2} : s_1' \xRightarrow{x} \wedge s_1][s_2 \xRightarrow{\sigma} s_1'][s_2' \\
\Rightarrow & (*\ \text{Lemma A.4.3 cases 1 and 4} \ *) \\
 & \exists s_1' \in Q_{s_1}, s_2' \in Q_{s_2} : s_1][s_2 \xRightarrow{\sigma} s_1'][s_2' \xRightarrow{x} \\
\Rightarrow & (*\ \text{Definition } out \text{ and } \textbf{after} \ *) \\
 & x \in out(s_1][s_2 \textbf{ after } \sigma)
\end{array}
$$

   The case for $x \in U_{s_2}$ is analogous, now using Lemma A.4.3 cases 2 and 5. Note that we make use of the fact that the transitions systems are convergent (see Lemma A.4.13 for a similar case).

   In case $x = \delta$ we obtain a similar result using Lemma A.4.3 case 6, combined with the knowledge that $x \in out(i_1 \textbf{ after } \sigma\lceil L^{\delta}_{i_1})$ and $x \in out(i_2 \textbf{ after } \sigma\lceil L^{\delta}_{s_2})$.

2. $\sigma\lceil L^{\delta}_{s_1} \in \mathit{Straces}(s_1)$ and $\sigma\lceil L^{\delta}_{s_2} \notin \mathit{Straces}(s_2)$. This means that the s_2 part of the parallel composition becomes chaotic. When $x \in U^{\delta}_{s_2}$ the chaotic process of s can perform this action, the non-chaotic part can always perform the corresponding input action if there is one. First

the case that $x \in U_{s_1}$, next $x \in U_{s_2}$ and we end with $x = \delta$. We know that $\sigma{\cdot}x\lceil L^\delta_{s_2}$ is a trace of i_2 ($\sigma{\cdot}x\lceil L^\delta_{s_2} = \sigma{\cdot}x\lceil L^\delta_{i_2}$). This means that the trace is executable by a transition system (a requirement we need for the lemmas we use). In the proof we refer to this system as r.

$$\begin{array}{ll}
 & \sigma\lceil L^\delta_{s_1} \in \mathit{Straces}(s_1) \wedge \sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2) \\
 & \wedge\, x \in \mathit{out}(i_1 \textbf{ after } \sigma\lceil L^\delta_{i_1}) \wedge \exists r \in \mathbf{LTS}(I_{s_2}, U_{s_2}) : r \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_2}} \\
\Rightarrow & (*\ i_1 \textbf{ ioco } s_1\ *) \\
 & \sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2) \wedge x \in \mathit{out}(s_1 \textbf{ after } \sigma\lceil L^\delta_{s_1}) \\
 & \wedge\, \exists r \in \mathbf{LTS}(I_{s_2}, U_{s_2}) : r \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_2}} \\
\Rightarrow & (*\ \text{Definition } \mathit{out} \text{ and } \textbf{after}\ *) \\
 & \sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2) \wedge s_1 \xRightarrow{\sigma\lceil L^\delta_{s_1}\cdot x} \\
 & \wedge\, \exists r \in \mathbf{LTS}(I_{s_2}, U_{s_2}) : r \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_2}} \\
\Rightarrow & (*\ \text{Lemma A.4.5 (projection)}\ *) \\
 & \sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2) \wedge s_1 \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_1}} \\
 & \wedge\, \exists r \in \mathbf{LTS}(I_{s_2}, U_{s_2}) : r \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_2}} \\
\Rightarrow & (*\ \text{Lemma A.4.16}\ *) \\
 & s_1][s_2 \xRightarrow{\sigma\cdot x} \\
\Rightarrow & (*\ \text{Definition } \mathit{out} \text{ and } \textbf{after}\ *) \\
 & x \in \mathit{out}(s_1][s_2 \textbf{ after } \sigma)
\end{array}$$

The case $x \in U_{s_2}$.

$$\begin{array}{ll}
 & \sigma\lceil L^\delta_{s_1} \in \mathit{Straces}(s_1) \wedge \sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2) \\
 & \wedge\, \exists r \in \mathbf{LTS}(I_{s_2}, U_{s_2}) : r \xRightarrow{\sigma\cdot x} \\
\Rightarrow & (*\ \text{Lemma A.4.17}\ *) \\
 & s_1][s_2 \xRightarrow{\sigma\cdot x} \\
\Rightarrow & (*\ \text{Definition } \mathit{out} \text{ and } \textbf{after} \text{ and clean up}\ *) \\
 & x \in \mathit{out}(s_1][s_2 \textbf{ after } \sigma)
\end{array}$$

The case $x = \delta$. This case is identical to the case for $x \in U_{s_1}$

3. $\sigma\lceil L^\delta_{s_1} \notin \mathit{Straces}(s_1)$ and $\sigma\lceil L^\delta_{s_2} \in \mathit{Straces}(s_2)$. This case is symmetrical with the previous one.

4. $\sigma\lceil L^\delta_{s_1} \notin \mathit{Straces}(s_1)$ and $\sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2)$. In this case both s_1 and s_2 are chaotic in the parallel composition and can do all necessary actions. The cases for $x \in U_r$, $x \in U_s$ and $x = \delta$ are all analogous.

$$\begin{array}{ll}
 & \sigma\lceil L^\delta_{s_1} \notin \mathit{Straces}(s_1) \wedge \sigma\lceil L^\delta_{s_2} \notin \mathit{Straces}(s_2) \wedge \exists r \in \mathbf{LTS}(I_{s_1}, U_{s_1}) : \\
 & r \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_1}} \wedge \exists t \in \mathbf{LTS}(I_{s_2}, U_{s_2}) : t \xRightarrow{\sigma\cdot x\lceil L^\delta_{s_2}} \\
\Rightarrow & (*\ \text{Lemma A.4.15}\ *) \\
 & s_1][s_2 \xRightarrow{\sigma\cdot x} \\
\Rightarrow & (*\ \text{Definitions } \mathit{out} \text{ and } \textbf{after}\ *) \\
 & x \in \mathit{out}(s_1][s_2 \textbf{ after } \sigma)
\end{array}$$

□

Appendix B

Proofs of Chapter 5: Atomic action refinement in MBT

B.1 Proofs Section 5.3: Trace refinement

We start with the trace refinement proofs, because we use these in the proofs on LTS refinement. This section centers around the proof of Proposition 5.3.11. The lemmas before it are used to complete its proof.

Lemma B.1.1 Let $\sigma \in L_\delta^*, \sigma' \in L_{r\delta}^*, r : L_\tau \rightarrow$ **FLTS**

$$\sigma' \in \sigma[r]_{rc} \Leftrightarrow \sigma \in \sigma'\langle r\rangle_{rc}$$

□

Proof

Only if: The case for $\sigma = \epsilon$ is straightforward, as $\epsilon[r]_{rc} = \epsilon\langle r\rangle_{rc} = \{\epsilon\}$.

Let $\sigma = \lambda_1 \cdots \lambda_n$ with $n \geq 1$ and $\forall 1 \leq i \leq n : \lambda_i \in L_\delta$.

$$\begin{array}{ll} & \sigma' \in (\lambda_1 \cdots \lambda_n)[r]_{rc} \\ \Rightarrow & (*\ \text{Definition 5.3.3 (complete trace refinement)}\ *) \\ & \sigma' \in \{\sigma_1 \cdots \sigma_n \mid \forall 1 \leq i \leq n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))\} \\ \Rightarrow & (*\ \text{Definition 5.3.7 (complete trace contraction)}\ *) \\ & \lambda_1 \cdots \lambda_n \in \sigma'\langle r\rangle_{rc} \\ \Rightarrow & (*\ \text{Premise: } \sigma = \lambda_1 \cdots \lambda_n\ *) \\ & \sigma \in \sigma'\langle r\rangle_{rc} \end{array}$$

If: The case for $\sigma = \epsilon$ is straightforward, as $\epsilon[r]_{rc} = \epsilon\langle r\rangle_{rc} = \{\epsilon\}$.

$$\begin{array}{ll}
 & \sigma \in \sigma'\langle r\rangle_{rc} \\
\Rightarrow & (*\ \text{Definition 5.3.7 (complete trace contraction)}\ \ *) \\
 & \sigma \in \{\lambda_1 \cdots \lambda_n \in L_\delta^* \mid \exists n > 0, \sigma_1, \ldots, \sigma_n \in L_{r\delta}^* : \sigma' = \sigma_1 \cdots \sigma_n \\
 & \wedge \forall 1 \leq i \leq n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \\
\Rightarrow & (*\ \text{Definition 5.3.3 (complete trace refinement)}\ \ *) \\
 & \sigma' \in \sigma[r]_{rc}
\end{array}$$

□

Lemma B.1.2 Let $\sigma \in L_\delta^*, \sigma' \in L_{r\delta}^*, r : L_\tau \to \mathbf{FLTS}$

$$\sigma' \in \sigma[r]_{inc} \Leftrightarrow \sigma \in \sigma'\langle r\rangle_{inc}$$

□

Proof

Only if: The case for $\sigma = \epsilon$ is vacuously true because $\epsilon[r]_{inc} = \emptyset$.

Let $\sigma = \lambda_1 \cdots \lambda_n$, with $n \geq 0$ and $\forall 1 \leq i \leq n : \lambda_i \in L_\delta$.

$$\begin{array}{ll}
 & \sigma' \in (\lambda_1 \cdots \lambda_n)[r]_{inc} \\
\Rightarrow & (*\ \text{Definition 5.3.4 (incomplete trace refinement)}\ \ *) \\
 & \sigma' \in \{\sigma_1 \cdots \sigma_n \mid \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)), \\
 & \sigma_n \in \mathit{XStraces}(r(\lambda_n))\} \\
\Rightarrow & (*\ \text{Definition 5.3.8 (incomplete trace contraction)}\ \ *) \\
 & \lambda_1 \cdots \lambda_n \in \sigma'\langle r\rangle_{inc} \\
\Rightarrow & (*\ \text{Premise: } \sigma = \lambda_1 \cdots \lambda_n\ \ *) \\
 & \sigma \in \sigma'\langle r\rangle_{inc}
\end{array}$$

If: The case for $\sigma = \epsilon$ is vacuously true, because $\epsilon\langle r\rangle_{inc} = \emptyset$.

$$\begin{array}{ll}
 & \sigma \in \sigma'\langle r\rangle_{inc} \\
\Rightarrow & (*\ \text{Definition 5.3.8 (incomplete trace contraction)}\ \ *) \\
 & \sigma \in \{\lambda_1 \cdots \lambda_n \in L_\delta^* \mid \exists n\, 0, \sigma_1, \ldots, \sigma_n : \sigma' = \sigma_1 \cdots \sigma_n \\
 & \wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \wedge \sigma_n \in \mathit{XStraces}(r(\lambda_n))\} \\
\Rightarrow & (*\ \text{Definition 5.3.4 (incomplete trace refinement)}\ \ *) \\
 & \sigma' \in \sigma[r]_{inc}
\end{array}$$

□

Proposition 5.3.11 Let $\sigma \in L_\delta^*, \sigma' \in L_{r\delta}^*, r : L_\tau \to \mathbf{FLTS}$

$$\sigma' \in \sigma[r] \Leftrightarrow \sigma \in \sigma'\langle r\rangle$$

□

Proof This proof follows directly from Lemma B.1.1 and Lemma B.1.2. □

The following lemmas are used in the proofs of the other sections. We have put them in this appendix, because they are about trace refinement.

Lemma B.1.3 Let $\sigma_1, \sigma_2 \in L_\delta^*, r : L_\tau \rightarrow \mathbf{FLTS}$

$$\{\sigma_1' \cdot \sigma_2' \mid \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \sigma_2[r]_{rc}\} = (\sigma_1 \cdot \sigma_2)[r]_{rc}$$

□

Proof

Subset ($\subseteq$) We identify the following cases:

$\sigma_1 = \epsilon$.

$$\begin{array}{ll} & \sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r]_{rc} \\ \Rightarrow & (*\ \text{Premise: } \sigma_1 = \epsilon \text{ and } \epsilon[r]_{rc} = \{\epsilon\}\ *) \\ & \sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r]_{rc} \wedge \sigma_1' \cdot \sigma_2' = \sigma_2' \wedge \sigma_1 \cdot \sigma_2 = \sigma_2 \\ \Rightarrow & (*\ \text{Logical reasoning}\ *) \\ & \sigma_1' \cdot \sigma_2' \in (\sigma_1 \cdot \sigma_2)[r]_{rc} \end{array}$$

$\sigma_2 = \epsilon$. This case is similar to case 1.

$\sigma_1, \sigma_2 \neq \epsilon$.

$$\begin{array}{ll} & \sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r]_{rc} \\ \Rightarrow & (*\ \text{Definition 5.3.3 (trace refinement) } \sigma_1, \sigma_2 \neq \epsilon\ *) \\ & \exists n \geq 1, \mu_1, \ldots, \mu_n \in L_\delta : \sigma_1 = \mu_1 \cdots \mu_n \\ & \wedge\, \sigma_1' \in \{\rho_1 \cdots \rho_m \mid \forall 1 \leq i \leq n : \rho_i \in \mathit{TXStraces}(r(\mu_i))\} \\ & \wedge\, \exists m \geq 1, \lambda_1, \ldots, \lambda_m \in L_\delta : \sigma_2 = \lambda_1 \cdots \lambda_m \\ & \wedge\, \sigma_2' \in \{\upsilon_1 \cdots \upsilon_m \mid \forall 1 \leq j \leq m : \upsilon_j \in \mathit{TXStraces}(r(\lambda_j))\} \\ \Rightarrow & (*\ \text{Set operations}\ *) \\ & \exists n, m \geq 1, \mu_1, \ldots, \mu_n, \lambda_1, \ldots, \lambda_n \in L_\delta : \sigma_1 = \mu_1 \cdots \mu_n \\ & \wedge\, \sigma_2 = \lambda_1 \cdots \lambda_m \wedge \sigma_1' \cdot \sigma_2' \in \{\rho_1 \cdots \rho_n \cdot \upsilon_1 \cdots \upsilon_n \mid \\ & \forall 1 \leq i \leq n : \rho_i \in \mathit{TXStraces}(r(\mu_i)), \\ & \forall 1 \leq j \leq m : \upsilon_j \in \mathit{TXStraces}(r(\lambda_j))\} \\ \Rightarrow & (*\ \text{Rewrite of } \sigma_1 = \mu_1 \cdots \mu_n \text{ and } \sigma_2 = \lambda_1 \cdots \lambda_n\ *) \\ & \exists k \geq 1, \mu_1, \ldots, \mu_k \in L_\delta : \sigma_1 \cdot \sigma_2 = \mu_1 \cdots \mu_k \\ & \wedge\, \sigma_1' \cdot \sigma_2' \in \{\rho_1 \cdots \rho_k \mid \forall 1 \leq i \leq k : \rho_i \in \mathit{TXStraces}(r(\mu_i))\} \\ \Rightarrow & (*\ \text{Definition 5.3.3 (complete trace refinement)}\ *) \\ & \sigma_1' \cdot \sigma_2' \in (\sigma_1 \cdot \sigma_2)[r]_{rc} \end{array}$$

Superset ($\supseteq$) We identify the following cases:

$\sigma_1 = \epsilon$

$$\begin{array}{ll} & \sigma \in (\sigma_1 \cdot \sigma_2)[r]_{rc} \\ \Rightarrow & (*\ \text{Premise } \sigma_1 = \epsilon\ *) \\ & \sigma \in \sigma_2[r]_{rc} \\ \Rightarrow & (*\ \text{Basic set operations}\ *) \\ & \sigma \in \{\sigma_2' \mid \sigma_2' \in \sigma_2[r]_{rc}\} \\ \Rightarrow & (*\ \text{Definition 5.3.3 (trace refinement), use premise } \sigma_1 = \epsilon \\ & \quad \text{and } \epsilon \in \epsilon[r]_{rc}\ *) \\ & \sigma \in \{\sigma_1' \cdot \sigma_2' \mid \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \sigma_2[r]_{rc}\} \end{array}$$

$\sigma_2 = \epsilon$. This case is similar to case 1.

$\sigma_1, \sigma_2 \neq \epsilon$. Assume that $\sigma_1 = \lambda_1 \cdots \lambda_n$ for some $n > 0$ and that $\sigma_2 = \mu_1 \cdots \mu_m$ for some $m > 0$ with $\forall m, n > 0 : \lambda_i, \mu_i \in L_\delta$.

$$\begin{array}{ll}
 & \sigma \in (\sigma_1 {\cdot} \sigma_2)[r]_{rc} \\
\Rightarrow & (*\ \text{Definition 5.3.3 (trace refinement)}\ *) \\
 & \sigma \in \{\rho_1 \cdots \rho_{m+n} \mid \forall 1 \leq i \leq n : \rho_i \in \mathit{TXStraces}(r(\lambda_i)), \\
 & \forall n+1 \leq i \leq m+n : \rho_i \in \mathit{TXStraces}(r(\mu_i))\} \\
\Rightarrow & (*\ \text{Definition 5.3.3 (trace refinement),} \\
 & \quad \text{use premise: } \sigma_1 = \lambda_1 \cdots \lambda_n,\ \sigma_2 = \mu_1 \cdots \mu_m\ *) \\
 & \sigma \in \{\sigma_1' {\cdot} \sigma_2' \mid \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \sigma_2[r]\}
\end{array}$$

□

Lemma B.1.4 Let $\sigma_1 \in L_\delta^*, \sigma_2 \in L_\delta^* \backslash \{\epsilon\}$

$$\{\sigma_1' {\cdot} \sigma_2' \mid \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \sigma_2[r]_{inc}\} = (\sigma_1 {\cdot} \sigma_2)[r]_{inc}$$

□

Proof

Subset ($\subseteq$) We identify the following cases:

- $\sigma_1 = \epsilon$

$$\begin{array}{ll}
 & \sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r]_{inc} \\
\Rightarrow & (*\ \text{Premise: } \sigma_1 = \epsilon \text{ and } \epsilon[r] = \{\epsilon\}\ \ *) \\
 & \sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r]_{inc} \wedge \sigma_1' {\cdot} \sigma_2' = \sigma_2' \wedge \sigma_1 {\cdot} \sigma_2 = \sigma_2 \\
\Rightarrow & (*\ \text{Logical reasoning}\ *) \\
 & \sigma_1' {\cdot} \sigma_2' \in (\sigma_1 {\cdot} \sigma_2)[r]_{inc}
\end{array}$$

- $\sigma_2 = \epsilon$. This case is ruled out explicitly.

- $\sigma_1, \sigma_2 \neq \epsilon$.

$$\begin{array}{ll}
 & \sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r]_{inc} \\
\Rightarrow & (*\ \text{Definition 5.3.4)}\ *) \\
 & \sigma_1' \in \sigma_1[r]_{rc} \wedge \exists n \geq 1, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma_2 = \lambda_1 \cdots \lambda_n \\
 & \wedge\, \sigma_2' \in \{\upsilon_1 \cdots \upsilon_n \mid \forall 1 \leq i < n : \upsilon_i \in \mathit{TXStraces}(r(\lambda_i)), \\
 & \upsilon_n \in \mathit{XStraces}(r(\lambda_n))\} \\
\Rightarrow & (*\ \text{Definition 5.3.3}\ *) \\
 & \exists m \geq 1, \mu_1 \cdots \mu_m \in L_\delta : \sigma_1 = \mu_1 \cdots \mu_m \wedge \sigma_1' \in \{\rho_1 \cdots \rho_m \mid \\
 & \forall 1 \leq i \leq m : \rho_1 \in \mathit{TXStraces}(r(\mu_i))\} \wedge \exists n \geq 1, \lambda_1 \cdots \lambda_n \in L_\delta : \\
 & \sigma_2 = \lambda_1 \cdots \lambda_n \wedge \sigma_2' \in \{\upsilon_1 \cdots \upsilon_n \mid \forall 1 \leq i < n : \\
 & \upsilon_i \in \mathit{TXStraces}(r(\lambda_i)), \upsilon_n \in \mathit{XStraces}(r(\lambda_n))\}
\end{array}$$

$\Rightarrow$ (* Set operations *)
$\exists n, m \geq 1, \mu_1 \cdots \mu_m, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma_1 = \mu_1 \cdots \mu_m$
$\wedge\, \sigma_2 = \lambda_1 \cdots \lambda_n \wedge \sigma_1' \cdot \sigma_2' \in \{\rho_1 \cdots \rho_m \cdot \upsilon_1 \cdots \upsilon_n \mid \forall 1 \leq i \leq m :$
$\rho_i \in \mathit{TXStraces}(r(\mu_i)), 1 \leq j < n : \upsilon_j \in \mathit{TXStraces}(r(\lambda_i)),$
$\upsilon_n \in \mathit{XStraces}(r(\lambda_n))\}$
$\Rightarrow$ (* Definition 5.3.4, logical reasoning *)
$\sigma_1' \cdot \sigma_2' \in (\sigma_1 \cdot \sigma_2)[r]_{inc}$

Superset ($\supseteq$) We identify the following cases:

- $\sigma_1 = \epsilon$
  $\sigma \in (\sigma_1 \cdot \sigma_2)[r]_{inc}$
  $\Rightarrow$ (* Premise: $\sigma_1 = \epsilon$ *)
  $\sigma \in \sigma_2[r]_{inc}$
  $\Rightarrow$ (* Basic set operations *)
  $\sigma \in \{\sigma_2' \mid \sigma_2' \in \sigma_2[r]_{inc}\}$
  $\Rightarrow$ (* Definition 5.3.3 (trace refinement), use premise: $\sigma_1 = \epsilon$ *)
  $\sigma \in \{\sigma_1' \cdot \sigma_2' \mid \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \sigma_2[r]_{inc}\}$
- $\sigma_2 = \epsilon$, this case is ruled out explicitly.
- $\sigma_1, \sigma_2 \neq \epsilon$. Assume that $\sigma_1 = \lambda_1 \cdots \lambda_n$ and that $\sigma_2 = \mu_1 \cdots \mu_m$ with $m, n > 0$.
  $\sigma \in (\sigma_1 \cdot \sigma_2)[r]_{inc}$
  $\Rightarrow$ (* Definition 5.3.4 (trace refinement) *)
  $\sigma \in \{\sigma_1' \cdots \sigma_{m+n}' \mid \forall 1 \leq i \leq n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)),$
  $\forall n+1 \leq i \leq m+n-1 : \sigma_i' \in \mathit{TXStraces}(r(\mu_i))$
  $\wedge\, \sigma_{m+n} \in \mathit{XStraces}(r(\mu_m))\}$
  $\Rightarrow$ (* Definition 5.3.3, Definition 5.3.4 (trace refinement), use premise: $\sigma_1 = \lambda_1 \cdots \lambda_n$, $\sigma_2 = \mu_1 \cdots \mu_m$ *)
  $\sigma \in \{\sigma_1' \cdot \sigma_2' \mid \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \sigma_2[r]_{inc}\}$

□

Lemma B.1.5 Let $\sigma_1, \sigma_2 \in L_\delta^*, r : L_\tau \to \mathbf{FLTS}$

$$\sigma_1' \in \sigma_1[r]_{rc} \wedge \sigma_2' \in \sigma_2[r] \Rightarrow \sigma_1' \cdot \sigma_2' \in (\sigma_1 \cdot \sigma_2)[r]$$

□

Proof Definition 5.3.5 gives two possibilities for σ_2' : $\sigma_2' \in \sigma_2[r]_{rc}$ or $\sigma_2' \in \sigma_2[r]_{inc}$. The first case is proven in Lemma B.1.3 and the second case in Lemma B.1.4.

□

Lemma B.1.6 Let $\sigma \in L_\delta^*, r : L_\tau \to \mathbf{FLTS}$

$$\sigma \in \sigma[r]_{rc}\langle r \rangle_{rc}$$

□

Proof Let $\sigma' \in \sigma[r]_{rc}$ (which always exists, because $\sigma[r]_{rc} \neq \emptyset$). Using Lemma B.1.1, this implies $\sigma \in \sigma'\langle r\rangle_{rc}$.

□

Lemma B.1.7 Let $\sigma \in L^*_{r\delta}, r : L_\tau \to \mathbf{FLTS}$

$$\sigma\langle r\rangle \neq \emptyset \Rightarrow \sigma \in \sigma\langle r\rangle_{rc}[r]_{rc}$$

□

Proof Let $\sigma' \in \sigma\langle r\rangle_{rc}$ (which always exists, because the premise is that $\sigma\langle r\rangle_{rc} \neq \emptyset$). Using Lemma B.1.1, this implies $\sigma \in \sigma'[r]_{rc}$.

□

B.2 Proofs Section 5.2: LTS refinement

One of the main results of LTS refinement is Theorem 5.3.12. Most of the material in this section are building blocks to construct its proof.

In the following lemma we show that the refined transition system can perform ϵ transitions, if the refinement transitions system can perform them.

Lemma B.2.1 Let $q_1, q_2 \in Q, \mu \in L_\tau, q_1', q_2' \in Q_{r(\mu)}$

$$q_1 \xrightarrow{\mu} q_2 \wedge q_1' \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q_2' \text{ implies } (q_2, q_1') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_2')$$

□

Proof We prove the following stronger case for some $n \geq 0$, by induction on n:

$$q_1 \xrightarrow{\mu} q_2 \wedge q_1' \xrightarrow{\tau^n}_{r(\mu)} q_2' \text{ implies } (q_2, q_1') \xrightarrow{\tau^n}_r (q_2, q_2')$$

Basic step: $n = 0$.

$$\begin{array}{ll} & q_1 \xrightarrow{\mu} q_2 \wedge q_1' \xrightarrow{\tau^0}_{r(\mu)} q_2' \\ \Rightarrow & (*\ \text{Definition 5.2.1 (LTS refinement } T_2) \quad *) \\ & q_1 \xrightarrow{\mu} q_2 \wedge (q_2, q_1') \xrightarrow{\tau^0}_r (q_2, q_2') \end{array}$$

Induction step: $\epsilon = \tau^n$ for some $n \geq 0$ and assume that the lemma holds for some $0 \leq i < n$.

$$
\begin{array}{ll}
 & q_1 \xrightarrow{\mu} q_2 \wedge q_1' \xrightarrow{\tau^{i+1}}_{r(\mu)} q_2' \\
\Rightarrow & (*\ \text{Definition} \rightarrow\ *) \\
 & q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3' \in Q_{r(\mu)} : q_1' \xrightarrow{\tau^i}_{r(\mu)} q_3' \xrightarrow{\tau}_{r(\mu)} q_2' \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3' \in Q_{r(\mu)} : q_1' \xrightarrow{\tau^i}_{r(\mu)} q_3' \xrightarrow{\tau}_{r(\mu)} q_2' \\
 & \wedge (q_2, q_1') \xrightarrow{\tau^i}_r (q_2, q_3') \\
\Rightarrow & (*\ \text{Definition 5.2.1 (LTS refinement } T_2)\ *) \\
 & \exists q_3' \in Q_{r(\mu)} : (q_2, q_1') \xrightarrow{\tau^i}_r (q_2, q_3') \wedge (q_2, q_3') \xrightarrow{\tau}_r (q_2, q_2')) \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & (q_2, q_1') \xrightarrow{\tau^{i+1}}_r (q_2, q_2')
\end{array}
$$

□

In the following lemma we show that the refined transition system can perform single action transitions (double arrow) if the refinement transition system can do so.

Lemma B.2.2 Let $q_1, q_2 \in Q, \mu \in L_\tau, q_1' \in Q_{r(\mu)}, q_2' \in Q_{r(\mu)} \backslash \{\mathsf{final}_{r(\mu)}\}, \sigma \in L^*_{r(\mu)\delta}$

$$q_1 \xrightarrow{\mu} q_2 \wedge q_1' \xRightarrow{\lambda}_{r(\mu)} q_2' \text{ implies } (q_2, q_1') \xRightarrow{\lambda}_r (q_2, q_2')$$

□

Proof We make a distinction between $\lambda \in L_{r(\mu)}$ and $\lambda = \delta$, beginning the proof with the former.

$$
\begin{array}{ll}
 & q_1 \xrightarrow{\mu} q_2 \wedge q_1' \xRightarrow{\lambda}_{r(\mu)} q_2' \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3, q_4' \in Q_{r(\mu)} : q_1' \xRightarrow{\epsilon}_{r(\mu)} q_3' \xrightarrow{\lambda}_{r(\mu)} q_4' \xRightarrow{\epsilon}_{r(\mu)} q_2' \\
\Rightarrow & (*\ \text{Lemma B.2.1}\ *) \\
 & q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3, q_4' \in Q_{r(\mu)} : q_1' \xRightarrow{\epsilon}_{r(\mu)} q_3' \xrightarrow{\lambda}_{r(\mu)} q_4' \xRightarrow{\epsilon}_{r(\mu)} q_2' \\
 & \wedge (q_2, q_1') \xRightarrow{\epsilon}_r (q_2, q_3') \wedge (q_2, q_4') \xRightarrow{\epsilon}_r (q_2, q_2') \\
\Rightarrow & (*\ \text{Definition 5.2.1 (LTS refinement } T_2)\ *) \\
 & (q_2, q_1') \xRightarrow{\epsilon}_r (q_2, q_3') \wedge (q_2, q_4') \xRightarrow{\epsilon}_r (q_2, q_2') \wedge (q_2, q_3') \xrightarrow{\lambda}_r (q_2, q_4') \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & (q_2, q_1') \xRightarrow{\lambda}_r (q_2, q_2')
\end{array}
$$

We continue with $\lambda = \delta$

$$
\begin{array}{ll}
 & q_1 \xrightarrow{\mu} q_2 \wedge q_1' \xRightarrow{\delta}_{r(\mu)} q_2' \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3, q_4' \in Q_{r(\mu)} : q_1' \xRightarrow{\epsilon}_{r(\mu)} q_3' \xrightarrow{\delta}_{r(\mu)} q_4' \xRightarrow{\epsilon}_{r(\mu)} q_2' \\
\Rightarrow & (*\ \text{Lemma B.2.1}\ *) \\
 & q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3, q_4' \in Q_{r(\mu)} : q_1' \xRightarrow{\epsilon}_{r(\mu)} q_3' \xrightarrow{\delta}_{r(\mu)} q_4' \xRightarrow{\epsilon}_{r(\mu)} q_2' \\
 & \wedge (q_2, q_1') \xRightarrow{\epsilon}_r (q_2, q_3') \wedge (q_2, q_4') \xRightarrow{\epsilon}_r (q_2, q_2')
\end{array}
$$

$\Rightarrow$ (∗ Definition δ ∗)
$q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3, q_4' \in Q_{r(\mu)} : q_1' \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q_3' \xrightarrow{\delta}_{r(\mu)} q_4' \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q_2'$
$\wedge (q_1, q_1') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_3') \wedge (q_2, q_4') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_2')$
$\wedge \forall \mu' \in U_{r(\mu)\tau} : q_3' \overset{\mu'}{\not\longrightarrow}_{r(\mu)} \wedge q_3' = q_4'$
$\Rightarrow$ (∗ Definition 5.2.1 (LTS refinement T_2) ∗)
$(q_1, q_1') \overset{\epsilon}{\Longrightarrow}_r q_2, q_3') \wedge (q_2, q_4') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_2')$
$\wedge \forall \mu' \in U_{r\tau} : (q_2, q_3') \overset{\mu'}{\not\longrightarrow}_r \wedge q_3' = q_4'$

This last step may need some explanation. q_3' cannot do an output action nor a τ action, in other words it can only do an input action. T_2 of Definition 5.2.1 shows that this means that (q_2, q_3') can also not do any output actions (or τ action) of $r(\mu)$. Furthermore, because the refinement transition systems have unique state spaces, and the only way to add intermediate transitions is via T_2, we know that (q_2, q_3) cannot do any action of $U_{r\tau}$.

It could be that an output action can be added via T_1, when q_3'/q_4' is a final state. This can only be the case when the transition $q_4' \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q_2'$ consists of zero τ steps; this means that $q_3' = q_4' = q_2'$. This case is explicitly ruled out, because the lemma is not applicable when q_2' is a final state.

$\Rightarrow$ (∗ Definition δ ∗)
$(q_1, q_1') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_3') \wedge (q_2, q_4') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_2') \wedge (q_2, q_3') \xrightarrow{\delta}_r (q_2, q_3') \wedge q_3' = q_4'$
$\Rightarrow$ (∗ Logical reasoning ∗)
$(q_1, q_1') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_3') \wedge (q_2, q_4') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_2') \wedge (q_2, q_3') \xrightarrow{\delta}_r (q_2, q_4')$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$(q_1, q_1') \overset{\delta}{\Longrightarrow}_r (q_2, q_2')$

□

In the following lemma we show that the refined transition system can perform any trace that the refinement transition system can perform.

Lemma B.2.3 Let $q_1, q_2 \in Q, \mu \in L_\tau, q_1' \in Q_{r(\mu)}, q_2' \in Q_{r(\mu)} \backslash \{\mathsf{final}_{r(\mu)}\}, \sigma \in L^*_{r(\mu)\delta}$

$$q_1 \xrightarrow{\mu} q_2 \wedge q_1' \overset{\sigma}{\Longrightarrow}_{r(\mu)} q_2' \text{ implies } (q_2, q_1') \overset{\sigma}{\Longrightarrow}_r (q_2, q_2')$$

□

Proof Proof by induction on the length of σ.

Basic step: $\sigma = \epsilon$. This case is proven in Lemma B.2.1.

Induction step: Let $\sigma = \sigma_1 \cdot \lambda_1$ with $\sigma_1 \in L^*_{r(\mu)\delta}$ and $\lambda_1 \in L_{r\delta}$.

$q_1 \xrightarrow{\mu} q_2 \wedge q_1' \overset{\sigma_1 \cdot \lambda_1}{\Longrightarrow}_{r(\mu)} q_2'$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3' \in Q_{r(\mu)} : q_1' \overset{\sigma_1}{\Longrightarrow}_{r(\mu)} q_3' \overset{\lambda_1}{\Longrightarrow}_{r(\mu)} q_2'$
$\Rightarrow$ (∗ Induction ∗)
$q_1 \xrightarrow{\mu} q_2 \wedge \exists q_3' \in Q_{r(\mu)} : q_1' \overset{\sigma_1}{\Longrightarrow}_{r(\mu)} q_3' \overset{\lambda_1}{\Longrightarrow}_{r(\mu)} q_2'$
$\wedge (q_2, q_1') \overset{\sigma_1}{\Longrightarrow}_r (q_2, q_3')$

$\Rightarrow$ (* Lemma B.2.2 *)
$(q_2, q_1') \xRightarrow{\sigma_1}_r (q_2, q_3') \wedge (q_2, q_3') \xRightarrow{\lambda_1}_r (q_2, q_2')$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$(q_2, q_1') \xRightarrow{\sigma_1 \cdot \lambda_1}_r (q_2, q_2')$
$\Rightarrow$ (* Premise: $\sigma = \sigma_1 \cdot \lambda_1$ *)
$(q_2, q_1') \xRightarrow{\sigma}_r (q_2, q_2')$

$\Box$

In the lemma's before we focused on T_2 transitions. In the following lemma's we add T_1 transitions, thus adding transitions starting in the start state of the refinement transition system.

The following lemma shows that the refined transition system can perform traces starting in the start state of the refinement transition system.

Lemma B.2.4 Let $q_1, q_2 \in Q, \mu \in L_\tau, q' \in Q_{r(\mu)}, q_2' \in Q_{r(\mu)} \backslash \{\mathsf{final}_{r(\mu)}\}, \lambda \in L_{r(\mu)}, \sigma \in L^*_{r(\mu)\delta}$

$$q_1 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda} q' \xRightarrow{\sigma} q_2' \text{ implies } \forall q_1' \in \mathsf{Final} : (q_1, q_1') \xRightarrow{\lambda \cdot \sigma}_r (q_2, q_2')$$

$\Box$

Proof
$q_1 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda}_{r(\mu)} q' \xRightarrow{\sigma}_{r(\mu)} q_2'$
$\Rightarrow$ (* Lemma B.2.3 *)
$q_1 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda}_{r(\mu)} q' \xRightarrow{\sigma}_{r(\mu)} q_2'$
$\wedge (q_2, q') \xRightarrow{\sigma}_r (q_2, q_2')$
$\Rightarrow$ (* Definition 5.2.1 (LTS refinement T_1) *)
$(q_2, q') \xRightarrow{\sigma}_r (q_2, q_2') \wedge \forall q_1' \in \mathsf{Final} : (q_1, q_1') \xrightarrow{\lambda}_r (q_2, q')$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\forall q_1' \in \mathsf{Final} : (q_1, q_1') \xRightarrow{\lambda \cdot \sigma}_r (q_2, q_2')$

$\Box$

The following lemma shows that the refined transition system can perform any trace that the refinement transition system can do between start and final state, as long as the trace does not end with δ.

Lemma B.2.5
Let $q_1, q_2 \in Q$, $\mu \in L_\tau$, $q' \in Q_{r(\mu)}$, $\lambda \in L_{r(\mu)}$, $\sigma \in L^*_{r(\mu)\delta} \backslash (L^*_{r(\mu)\delta} \cdot \delta)$

$$q_1 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda} q' \xRightarrow{\sigma} \mathsf{final}_{r(\mu)} \text{ implies}$$
$$\forall q_1' \in \mathsf{Final} : (q_1, q_1') \xRightarrow{\lambda \cdot \sigma}_r (q_2, \mathsf{final}_{r(\mu)})$$

$\Box$

Proof We want to apply Lemma B.2.4, but this lemma is not directly applicable because we use the final state $\mathsf{final}_{r(\mu)}$. Therefore we refer to the state before $\mathsf{final}_{r(\mu)}$. Let $\sigma = \sigma_1 \cdot \mu_1$. Note that we assume that σ does not end with δ (i.e., $\mu_1 \in L_{r(\mu)\tau}$.

$\Rightarrow$ (∗ Definition $\Longrightarrow$ and premise ∗)
$q_1 \xrightarrow{\mu} q_2 \wedge \exists q_1' \in Q_{r(\mu)} : r(\mu) \xrightarrow{\lambda}_{r(\mu)} q' \xRightarrow{\sigma_1}_{r(\mu)} q_1' \xrightarrow{\mu_1}_{r(\mu)} \mathsf{final}_{r(\mu)}$
$\Rightarrow$ (∗ Definition 4.6.1 constraint 5: no outgoing transitions in final ∗)
$q_1 \xrightarrow{\mu} q_2 \wedge \exists q_1' \in Q_{r(\mu)} : r(\mu) \xrightarrow{\lambda}_{r(\mu)} q' \xRightarrow{\sigma_1}_{r(\mu)} q_1' \xrightarrow{\mu_1}_{r(\mu)} \mathsf{final}_{r(\mu)}$
$\wedge\, q_1' \notin \mathsf{Final}$
$\Rightarrow$ (∗ Lemma B.2.4 ∗)
$q_1 \xrightarrow{\mu} q_2 \wedge \exists q_1' \in Q_{r(\mu)} : q_1' \xrightarrow{\mu_1}_{r(\mu)} \mathsf{final}_{r(\mu)}$
$\wedge\, \forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\lambda\cdot\sigma_1}_r (q_2, q_1')$
$\Rightarrow$ (∗ Definition 5.2.1 (LTS refinement T_2) ∗)
$\forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\lambda\cdot\sigma_1}_r (q_2, q_1') \wedge (q_2, q_1') \xrightarrow{\mu_1}_r (q_2, \mathsf{final}_{r(\mu)})$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\lambda\cdot\sigma_1\cdot\mu_1}_r (q_2, \mathsf{final}_{r(\mu)})$
$\Rightarrow$ (∗ Premise: $\sigma = \sigma_1\cdot\mu_1$ ∗)
$\forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\lambda\cdot\sigma}_r (q_2, \mathsf{final}_{r(\mu)})$

□

In the following lemma we show the relation between abstract actions μ and the possible completely refined traces in the refined system.

Lemma B.2.6 Let $q, q_1 \in Q, q' \in \mathsf{Final}, \mu \in L_\tau$

$$q \xrightarrow{\mu} q_1 \wedge \sigma \in \mu[r]_{rc} \Rightarrow (q, q') \xRightarrow{\sigma}_r (q_1, \mathsf{final}_{r(\mu)})$$

□

Proof From the definition of LTS refinement (Definition 5.2.1) we see that there are two ways to add transitions: T_1 and T_2. As $q' \in \mathsf{Final}$ we know that only T_1 is applicable in state (q, q'). We identify the following cases:

- $\mu = \tau$. $r(\tau) \xrightarrow{\tau} \mathsf{final}_{r(\tau)}$ is the only transition in $r(\tau)$. This means that $\tau[r]_{rc} = \{\epsilon\}$.

  $q \xrightarrow{\tau} q_1 \wedge r(\tau) \xrightarrow{\tau} \mathsf{final}_{r(\tau)}$
  $\Rightarrow$ (∗ Definition 5.2.1 (LTS refinement) ∗)
  $(q, q') \xrightarrow{\tau}_r (q_1, \mathsf{final}_{r(\tau)})$

- $\mu \neq \tau$. Because $\sigma \in \mu[r]_{rc}$ we know that σ does not start or end with δ. Let $\sigma = \lambda_1 \cdots \lambda_n$ for some $n \geq 1$ and $\forall 1 \leq i \leq n : \lambda_i \in L_{r(\mu)\delta}$.

  $q \xrightarrow{\mu} q_1 \wedge (\lambda_1 \cdots \lambda_n) \in \mu[r]_{rc}$
  $\Rightarrow$ (∗ Definition 5.3.3 (complete trace refinement) ∗)
  $q \xrightarrow{\mu} q_1 \wedge (\lambda_1 \cdots \lambda_n) \in \mathit{TXStraces}(r(\mu))$
  $\Rightarrow$ (∗ Definition 5.3.1 ($\mathit{TXStraces}$) ∗)
  $q \xrightarrow{\mu} q_1 \wedge r(\mu) \xRightarrow{\lambda_1\cdots\lambda_n} \mathsf{final}_{r(\mu)} \wedge (\lambda_1 \cdots \lambda_n) \notin (\delta\cdot L^*_{r(\mu)\delta} \cup L_{r(\mu)\delta}\cdot\delta)$

$\Rightarrow$ (* Logical reasoning *)
$q \xrightarrow{\mu} q_1 \wedge r(\mu) \xRightarrow{\lambda_1 \cdots \lambda_n} \mathsf{final}_{\mu[r]} \wedge \lambda_1, \lambda_n \neq \delta$
$\Rightarrow$ (* Constraints 1,2,3 in Definition 4.6.1 (σ does not start with a τ action) together with definition $\Longrightarrow$ and $\longrightarrow$ *)
$q \xrightarrow{\mu} q_1 \wedge \exists q_1' \in Q_{r(\mu)} : r(\mu) \xrightarrow{\lambda_1}_r q_1' \xRightarrow{\lambda_2 \cdots \lambda_n}_r \mathsf{final}_{r(\mu)} \wedge \lambda_1, \lambda_n \neq \delta$
$\Rightarrow$ (* Lemma B.2.5, note that $\lambda_1, \lambda_n \neq \delta$ *)
$\forall q' \in \mathsf{Final} : (q, q') \xRightarrow{\lambda_1 \cdots \lambda_n}_r (q_1, \mathsf{final}_{r(\mu)})$
$\Rightarrow$ (* Premise: $\sigma = \lambda_1 \cdots \lambda_n$ *)
$\forall q' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma}_r (q_1, \mathsf{final}_{r(\mu)})$

□

In the following lemma we show the relation between abstract actions μ and the possible incompletely refined traces in the refined system.

Lemma B.2.7 Let $q, q_1 \in Q, q' \in \mathsf{Final}, \mu \in L_\tau$

$$q \xrightarrow{\mu} q_1 \wedge \sigma \in \mu[r]_{inc} \Rightarrow (q, q') \xRightarrow{\sigma}_r$$

□

Proof We identify the following cases:

- $\mu = \tau$. Because $\tau[r]_{inc} = \emptyset$, the lemma is vacuously true for this case.
- $\mu \neq \tau$. Because $\sigma \in \mu[r]_{inc}$ we know that σ does not start with δ, furthermore we know that σ does not end in a final state in $r(\mu)$. Let $\sigma = \lambda_1 \cdots \lambda_n$ for some $n \geq 1$ and $\forall 1 \leq i \leq n : \lambda_i \in L_{r(\mu)\delta}$.

  $q \xrightarrow{\mu} q_1 \wedge (\lambda_1 \cdots \lambda_n) \in \mu[r]_{inc}$
  $\Rightarrow$ (* Definition 5.3.4 (incomplete trace refinement) *)
  $q \xrightarrow{\mu} q_1 \wedge (\lambda_1 \cdots \lambda_n) \in \mathit{XStraces}(r(\mu))$
  $\Rightarrow$ (* Definition 5.3.2 (*XStraces*) *)
  $q \xrightarrow{\mu} q_1 \wedge \exists q' \in Q_{r(\mu)} \backslash \{\mathsf{final}_{r(\mu)}\} : r(\mu) \xRightarrow{\lambda_1 \cdots \lambda_n} q'$
  $\wedge (\lambda_1 \cdots \lambda_n) \notin (\delta \cdot L^*_{r(\mu)\delta} \cup \{\epsilon\})$
  $\Rightarrow$ (* Logical reasoning *)
  $q \xrightarrow{\mu} q_1 \wedge \exists q' \in Q_{r(\mu)} \backslash \{\mathsf{final}_{r(\mu)}\} : r(\mu) \xRightarrow{\lambda_1 \cdots \lambda_n} q' \wedge \lambda_1 \neq \delta$
  $\Rightarrow$ (* Constraints 1,2,3 in Definition 4.6.1 (σ does not start with a τ action) together with definition $\Longrightarrow$ and $\longrightarrow$ *)
  $q \xrightarrow{\mu} q_1 \wedge \exists q' \in Q_{r(\mu)} \backslash \{\mathsf{final}_{r(\mu)}\}, q_1' \in Q_{r(\mu)} : r(\mu) \xrightarrow{\lambda_1} q_1' \xRightarrow{\lambda_2 \cdots \lambda_n} q'$
  $\wedge \lambda_1 \neq \delta$
  $\Rightarrow$ (* Lemma B.2.4 *)
  $\forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\lambda_1 \cdots \lambda_n}_r (q_1, q')$
  $\Rightarrow$ (* Definition $\Longrightarrow$ *)
  $\forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\lambda_1 \cdots \lambda_n}_r$
  $\Rightarrow$ (* Premise: $\sigma = \lambda_1 \cdots \lambda_n$ *)
  $\forall q_2' \in \mathsf{Final} : (q_1, q_2') \xRightarrow{\sigma}_r$

□

We introduce a definition for the concept that a trace, or more precise an execution fragment in a refined system does not encounter intermediate states where the second state element is in Final. We call this property *final state clean* of *fsclean* for short.

Definition B.2.8 Let $(q_0, q_0'), (q, q') \in Q_r, \sigma = \lambda_1 \cdots \lambda_n$ for some $n \geq 1$ and $\forall 1 \leq i \leq n : \lambda_i \in L_{r\tau\delta}$ (we use the notation $(q_n, q_n') = (q, q')$).

$\mathit{fsclean}[(q_0, q_0') \xrightarrow{\lambda_1}_r \cdots \xrightarrow{\lambda_n}_r (q, q')] =_{\text{def}}$
$\forall 1 \leq i < n : (q_{i-1}, q_{i-1}') \xrightarrow{\lambda_i}_r (q_i, q_i') \Rightarrow q_i' \notin \mathsf{Final}$

We lift this notation in a straightforward manner to the double arrow notation.

The following lemmas are the building blocks for Lemma B.2.12. We start with single actions, continue with ϵ and end with traces.

Lemma B.2.9 Let $(q_0, q_0'), (q, q') \in Q_r, q_0' \notin \mathsf{Final}, \lambda \in L_{r\delta}$

$$\mathit{fsclean}[(q_0, q_0') \xrightarrow{\lambda}_r (q, q')] \Rightarrow \exists \mu \in L_\tau : q_0' \xrightarrow{\lambda}_{r(\mu)} q' \wedge q_0 = q$$

□

Proof We identify the following cases:

- $\lambda \in L_r$

  $\Rightarrow$ (∗ Definition 5.2.1 (LTS refinement T_2) ∗)
  $\exists \mu \in L_\tau : q_0' \xrightarrow{\lambda}_{r(\mu)} q' \wedge q_0 = q$

- $\lambda = \delta$

  $\Rightarrow$ (∗ Definition δ ∗)
  $\mathit{fsclean}[(q_0, q_0') \xrightarrow{\delta}_r (q, q')]$
  $\wedge \forall \mu \in U_{r\tau} : (q_0, q_0') \overset{\mu'}{\not\rightarrow}_r \wedge (q_0, q_0') = (q, q')$

  $\Rightarrow$ (∗ Logical reasoning using Definition 5.2.1 (LTS refinement T_2) and Definition 4.6.1: 1,2,3 ∗)
  $\exists \mu \in L_\tau : (\forall \mu' \in U_{r(\mu')} \cup \{\tau\} : q_0' \overset{\mu'}{\not\rightarrow}_{r(\mu)}) \wedge (q_0, q_0') = (q, q')$

  $\Rightarrow$ (∗ Definition δ ∗)
  $\exists \mu \in L_\tau : q_0' \xrightarrow{\delta}_{r(\mu)} q_0' \wedge (q_0, q_0') = (q, q')$

  $\Rightarrow$ (∗ Logical reasoning ∗)
  $\exists \mu \in L_\tau : q_0' \xrightarrow{\delta}_{r(\mu)} q' \wedge q_0 = q$

□

Lemma B.2.10 Let $(q_0, q_0'), (q, q') \in Q_r, q_0' \notin \mathsf{Final}$

$$\mathit{fsclean}[(q_0, q_0') \overset{\epsilon}{\Longrightarrow}_r (q, q')] \Rightarrow \exists \mu \in L_\tau : q_0' \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q$$

□

Proof We prove the following stronger statement for some $n \geq 0$:

$$\mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^n}_r (q, q')] \Rightarrow \exists \mu \in L_\tau : q_0' \xrightarrow{\tau^n}_{r(\mu)} q' \wedge q_0 = q$$

Proof by induction on n.

Basic step: $n = 0$.

$$\begin{array}{ll}
 & \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^0}_r (q, q')] \\
\Rightarrow & (*\ \text{Definition } \tau^0\ *) \\
 & \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^0}_r (q, q')] \wedge (q_0, q_0') = (q, q') \\
\Rightarrow & (*\ \text{Logical reasoning, using the definition of } \tau^0\ *) \\
 & \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^0}_r (q, q')] \wedge q_0' \xrightarrow{\tau^0} q' \wedge q_0 = q \\
\Rightarrow & (*\ \text{Logical reasoning, using Definition 5.2.1 (lts refinement } T_2\text{);} \\
 & \quad \text{states of refinement transition systems are unique } *) \\
 & \exists \mu \in L_\tau : q_0' \xrightarrow{\tau^0}_{r(\mu)} q' \wedge q_0 = q
\end{array}$$

Induction step: $n = j + 1$. Assume that the lemma holds for j

$$\begin{array}{ll}
 & \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^j \cdot \tau}_r (q, q')] \\
\Rightarrow & (*\ \text{Definition } \longrightarrow\ *) \\
 & \exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^j}_r (q_1, q_1') \xrightarrow{\tau}_r (q, q')] \\
\Rightarrow & (*\ \text{Induction } *) \\
 & \exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^j}_r (q_1, q_1') \xrightarrow{\tau}_r (q, q')] \\
 & \wedge \exists \mu \in L_\tau : q_0' \xrightarrow{\tau^j}_{r(\mu)} q_1' \wedge q_0 = q_1 \\
\Rightarrow & (*\ \text{Definition 5.2.1 (LTS refinement } T_2\text{). } *) \\
 & \exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^j}_r (q_1, q_1') \xrightarrow{\tau}_r (q, q')] \\
 & \wedge \exists \mu \in L_\tau : q_0' \xrightarrow{\tau^j}_{r(\mu)} q_1' \wedge q_0 = q_1 \wedge \exists \mu' \in L_\tau : q_1' \xrightarrow{\tau}_{r(\mu')} q' \wedge q_1 = q \\
\Rightarrow & (*\ \text{Logical reasoning using Definition 5.2.1 (LTS refinement):} \\
 & \quad \mu = \mu'\text{, because states of refinement transition systems} \\
 & \quad \text{are unique. } *) \\
 & \exists \mu \in L_\tau : q_0' \xrightarrow{\tau^j}_{r(\mu)} q_1' \xrightarrow{\tau}_{r(\mu)} q' \wedge q_0 = q \\
\Rightarrow & (*\ \text{Definition } \longrightarrow\ *) \\
 & \exists \mu \in L_\tau : q_0' \xrightarrow{\tau^j \cdot \tau}_{r(\mu)} q' \wedge q_0 = q
\end{array}$$

□

Lemma B.2.11 Let $(q_0, q'_0), (q, q') \in Q_r, q'_0 \notin \mathsf{Final}, \lambda \in L_{r\delta}$

$$fsclean[(q_0, q'_0) \overset{\lambda}{\Longrightarrow}_r (q, q')] \Rightarrow \exists \mu \in L_\tau : q'_0 \overset{\lambda}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q$$

□

Proof

$fsclean[(q_0, q'_0) \overset{\lambda}{\Longrightarrow}_r (q, q')]$

$\Rightarrow$ $(*$ Definition $\Longrightarrow$, using fsclean definition $*)$
$\exists (q_1, q'_1), (q_2, q'_2) \in Q_r :$
$fsclean[(q_0, q'_0) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1) \xrightarrow{\lambda}_r (q_2, q'_2) \overset{\epsilon}{\Longrightarrow}_r (q, q')] \wedge q'_1, q'_2 \notin \mathsf{Final}$

$\Rightarrow$ $(*$ Lemma B.2.10 $*)$
$\exists (q_1, q'_1), (q_2, q'_2) \in Q_r :$
$fsclean[(q_0, q'_0) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1) \xrightarrow{\lambda}_r (q_2, q'_2) \overset{\epsilon}{\Longrightarrow}_r (q, q')] \wedge q'_1 \notin \mathsf{Final}$
$\exists \mu_1 \in L_\tau : q'_0 \overset{\epsilon}{\Longrightarrow}_{r(\mu_1)} q'_1 \wedge q_0 = q_1 \wedge \exists \mu_2 \in L_\tau : q'_2 \overset{\epsilon}{\Longrightarrow}_{r(\mu_2)} q' \wedge q_2 = q$

$\Rightarrow$ $(*$ Lemma B.2.9 $*)$
$\exists (q_1, q'_1), (q_2, q'_2) \in Q_r :$
$\exists \mu_1 \in L_\tau : q'_0 \overset{\epsilon}{\Longrightarrow}_{r(\mu_1)} q'_1 \wedge q_0 = q_1$
$\wedge \exists \mu_2 \in L_\tau : q'_2 \overset{\epsilon}{\Longrightarrow}_{r(\mu_2)} q' \wedge q_2 = q$
$\wedge \exists \mu_3 \in L_\tau : q'_1 \xrightarrow{\lambda}_{r(\mu_3)} q'_2 \wedge q_1 = q_2$

$\Rightarrow$ $(*$ States of refinement transition systems are unique.
Therefore $\mu_1 = \mu_2 = \mu_3$ $*)$
$\exists (q_1, q'_1), (q_2, q'_2) \in Q_r, \mu \in L_\tau :$
$q'_0 \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q'_1 \xrightarrow{\lambda}_{r(\mu)} q'_2 \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q$

$\Rightarrow$ $(*$ Definition $\Longrightarrow$ $*)$
$\exists \mu \in L_\tau : q'_0 \overset{\lambda}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q$

□

The following lemma shows that in a refined transition system, traces that are fsclean consist entirely of T_2 transitions when the second element of the starting state is not in Final.

Lemma B.2.12 Let $(q_0, q'_0), (q, q') \in Q_r, q'_0 \notin \mathsf{Final}, \sigma \in L^*_{r\delta}$

$$fsclean[(q_0, q'_0) \overset{\sigma}{\Longrightarrow}_r (q, q')] \Rightarrow \exists \mu \in L_\tau : q'_0 \overset{\sigma}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q$$

□

Proof To understand this proof it is important to realize that the state spaces of refinement transition system are distinct. This makes it possible to uniquely identify states in a refinement transition system. As a result we can 'connect' incoming and outgoing transitions to states in refinement transition systems.
Proof by induction on the length of σ

Basic step: $\sigma = \epsilon$. This step follows from Lemma B.2.10.

Induction step: $\sigma = \sigma' \cdot \lambda$ and assume that the lemma holds for σ'.

$$\begin{array}{ll}
 & \mathit{fsclean}[(q_0, q_0') \xrightarrow{\sigma'\cdot\lambda}\!\!\!\!\Rightarrow_r (q, q')] \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \overset{\sigma'}{\Longrightarrow}_r (q_1, q_1') \overset{\lambda}{\Longrightarrow}_r (q, q')] \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \overset{\sigma'}{\Longrightarrow}_r (q_1, q_1') \overset{\lambda}{\Longrightarrow}_r (q, q')] \\
 & \wedge\, \exists \mu \in L_\tau : q_0' \overset{\sigma'}{\Longrightarrow}_{r(\mu)} q_1' \wedge q_0 = q_1 \\
\Rightarrow & (*\ \text{Lemma B.2.11, } q_1' \notin \mathsf{Final}\text{, because of } \mathit{fsclean}[]\ *) \\
 & \exists (q_1, q_1') \in Q_r, \mu \in L_\tau : q_0' \overset{\sigma'}{\Longrightarrow}_{r(\mu)} q_1' \wedge q_0 = q_1 \\
 & \wedge\, \exists \mu_1 \in L_\tau : q_1' \overset{\lambda}{\Longrightarrow}_{r(\mu_1)} q' \wedge q_1 = q \\
\Rightarrow & (*\ \text{States of refinement transition systems are unique.} \\
 & \quad \text{Therefore } \mu = \mu_1\ *) \\
 & \exists \mu \in L_\tau : q_0' \overset{\sigma'}{\Longrightarrow}_{r(\mu)} q_1' \overset{\lambda}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists \mu \in L_\tau : q_0' \overset{\sigma'\cdot\lambda}{\Longrightarrow}_{r(\mu)} q' \wedge q_0 = q
\end{array}$$

□

The following lemmas are the building blocks for Lemma B.2.16. That lemma is similar to Lemma B.2.12 above, but describes the case that q_0' (the second element of the first state) is a member of Final.

Note that although we do not always make it explicit, in the proofs we do make use of the fact that final states do not have outgoing transitions.

Lemma B.2.13 Let $(q_0, q_0'), (q, q') \in Q_r$ with $q_0' \in \mathsf{Final}$
$\mathit{fsclean}[(q_0, q_0') \overset{\epsilon}{\Longrightarrow}_r (q, q')]$

$$\Rightarrow ((q_0, q_0') = (q, q') \vee (q_0 \xrightarrow{\tau} q \wedge r(\tau) \xrightarrow{\tau} q' \wedge q' = \mathsf{final}_{r(\tau)}))$$

□

Proof

$$\begin{array}{ll}
 & \mathit{fsclean}[(q_0, q_0') \overset{\epsilon}{\Longrightarrow}_r (q, q')] \\
\Rightarrow & (*\ \text{Definition } \epsilon\ *) \\
 & \exists n \geq 0 : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau^n}_r (q, q')]
\end{array}$$

According to Definition 5.2.1 (LTS refinement) there are a couple of possibilities. One is that $n = 0$ and that $(q_0, q_0') = (q, q')$, thus fulfilling the first part of the or-clause. Because of Definition 4.6.1: constraints 1,2 and 3 only the refinement transition system τ is allowed to start with τ, this is the second possibility.

$$\begin{array}{ll}
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists n \geq 0, (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau}_r (q_1, q_1') \xrightarrow{\tau^{n-1}} (q, q')] \\
\Rightarrow & (*\ \text{Definition 5.2.1 (LTS refinement } T_1)\ *) \\
 & \exists n \geq 0, (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\tau}_r (q_1, q_1') \xrightarrow{\tau^{n-1}} (q, q')] \\
 & \wedge\, q_0 \xrightarrow{\tau} q_1 \wedge r(\tau) \xrightarrow{\tau} q_1' \wedge q_1' = \mathsf{final}_{r(\tau)} \\
\Rightarrow & (*\ \text{Logical reasoning, fsclean property, } q_1' = \mathsf{final}_{r(\tau)} : n = 1\ *) \\
 & q_0 \xrightarrow{\tau} q \wedge r(\tau) \xrightarrow{\tau} q' \wedge q' = \mathsf{final}_{r(\tau)}
\end{array}$$

□

Lemma B.2.14 Let $\lambda \in L_r, (q_0, q_0'), (q, q') \in Q_r$ with $q_0' \in \mathsf{Final}$

$$\mathit{fsclean}[(q_0, q_0') \overset{\lambda}{\Longrightarrow}_r (q, q')] \Rightarrow \exists \mu \in L : q_0 \xrightarrow{\mu} q \wedge r(\mu) \overset{\lambda}{\Longrightarrow} q'$$

□

Proof

$\mathit{fsclean}[(q_0, q_0') \overset{\lambda}{\Longrightarrow}_r (q, q')]$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists (q_1, q_1'), (q_2, q_2') \in Q_r :$
$\mathit{fsclean}[(q_0, q_0') \overset{\epsilon}{\Longrightarrow}_r (q_1, q_1') \xrightarrow{\lambda}_r (q_2, q_2') \overset{\epsilon}{\Longrightarrow}_r (q, q')]$

According to Lemma B.2.13, either $(q_0, q_0') = (q_1, q_1')$ or $q_1' = \mathsf{final}_{r(\tau)}$. The later leads to a contradiction, because of the fsclean property.

$\Rightarrow$ (∗ Lemma B.2.13 ∗)
$\exists (q_2, q_2') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\lambda}_r (q_2, q_2') \overset{\epsilon}{\Longrightarrow}_r (q, q')]$
$\Rightarrow$ (∗ Definition 5.2.1 (LTS refinement T_1) ∗)
$\exists (q_2, q_2') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xrightarrow{\lambda}_r (q_2, q_2') \overset{\epsilon}{\Longrightarrow}_r (q, q')]$
$\wedge \exists \mu \in L : q_0 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda} q_2'$
$\Rightarrow$ (∗ Lemma B.2.12 ∗)
$\exists (q_2, q_2') \in Q_r, \mu \in L : q_0 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda} q_2'$
$\wedge \exists \mu' \in L_\tau : q_2' \overset{\epsilon}{\Longrightarrow}_{r(\mu')} q' \wedge q_2 = q$
$\Rightarrow$ (∗ Definition 5.2.1: states of refinement transition systems are unique ∗)
$\exists (q_2, q_2') \in Q_r, \exists \mu \in L : q_0 \xrightarrow{\mu} q_2 \wedge r(\mu) \xrightarrow{\lambda}_{r(\mu)} q_2' \overset{\epsilon}{\Longrightarrow}_{r(\mu)} q'$
$\wedge q_2 = q$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ with logical reasoning: $q_2 = q$ ∗)
$\exists \mu \in L : q_0 \xrightarrow{\mu} q \wedge r(\mu) \overset{\lambda}{\Longrightarrow}_{r(\mu)} q$

□

Lemma B.2.15 Let $(q_0, q_0'), (q, q') \in Q_r$ with $q_0' \in \mathsf{Final}$

$$\mathit{fsclean}[(q_0, q_0') \overset{\delta}{\Longrightarrow}_r (q, q')] \Rightarrow q_0 \xrightarrow{\delta} q \wedge (q_0, q_0') = (q, q')$$

□

Proof

$\mathit{fsclean}[(q_0, q_0') \overset{\delta}{\Longrightarrow}_r (q, q')]$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists (q_1, q_1'), (q_2, q_2') \in Q_r :$
$\mathit{fsclean}[(q_0, q_0') \overset{\epsilon}{\Longrightarrow}_r (q_1, q_1') \xrightarrow{\delta}_r (q_2, q_2') \overset{\epsilon}{\Longrightarrow}_r (q, q')]$

According to Lemma B.2.13, either $(q_0, q_0') = (q_1, q_1')$ or $q_1' = \mathsf{final}_{r(\tau)}$. The last case leads to a contradiction, because of the fsclean property.

$\Rightarrow$ (* Lemma B.2.13 *)
$\exists(q_2,q_2') \in Q_r : \mathit{fsclean}[(q_0,q_0') \xrightarrow{\delta}_r (q_2,q_2') \overset{\epsilon}{\Longrightarrow}_r (q,q')]$
$\Rightarrow$ (* Definition δ *)
$\exists(q_2,q_2') \in Q_r : \mathit{fsclean}[(q_0,q_0') \xrightarrow{\delta}_r (q_2,q_2') \overset{\epsilon}{\Longrightarrow}_r (q,q')]$
$\wedge\, (q_0,q_0') = (q_2,q_2') \wedge \forall\lambda \in U_{r\tau} : (q_0,q_0') \not\xrightarrow{\lambda}_r$
$\Rightarrow$ (* Definition B.2.8 (fsclean):
($q_0' = q_2' \in \mathsf{Final}$, therefore $(q_2,q_2') = (q,q')$ *)
$(q_0,q_0') = (q,q') \wedge \forall\lambda \in U_{r\tau} : (q_0,q_0') \not\xrightarrow{\lambda}_r$
$\Rightarrow$ (* Definition 5.2.1 (LTS refinement T_1),
Definition 4.6.1: constraints 1,2,3 *)
$(q_0,q_0') = (q,q') \wedge \forall\lambda' \in U_\tau : q_0 \not\xrightarrow{\lambda'}$
$\Rightarrow$ (* Definition δ *)
$(q_0,q_0') = (q,q') \wedge q_0 \xrightarrow{\delta} q_0$
$\Rightarrow$ (* Logical reasoning *)
$q_0 \xrightarrow{\delta} q \wedge (q_0,q_0') = (q,q')$

□

Lemma B.2.16 Let $\sigma \in L_{r\delta}^*\backslash\{\epsilon,\delta\}, (q_1,q_1'), (q,q') \in Q_r$ with $q_1' \in \mathsf{Final}$

$$\mathit{fsclean}[(q_1,q_1') \overset{\sigma}{\Longrightarrow}_r (q,q')] \Rightarrow \exists\lambda \in L_{\tau\delta} : q_1 \xrightarrow{\lambda} q \wedge r(\lambda) \overset{\sigma}{\Longrightarrow} q'$$

□

Proof This follows directly from the way we construct refined transition systems: Definition 5.2.1. There are two rules to add transitions in the refined system: T_1 and T_2. T_1 is only applicable for the first transition of the refinement transition system (when $q_1' \in \mathsf{Final}$). For all other transitions we are sure that the second part of the state tuple is not in Final because of the fsclean property (Definition B.2.8). Therefore T_2 is applicable for those transitions.

Suppose $\sigma = \lambda_1{\cdot}\sigma_1$ for some $\lambda_1 \in L_{r\delta}$ and $\sigma_1 \in L_{r\delta}^*$. We identify the following cases:

- $\lambda_1 \in L$
  $\mathit{fsclean}[(q_1,q_1') \overset{\lambda_1\cdot\sigma_1}{\Longrightarrow}_r (q,q')]$
  $\Rightarrow$ (* Definition $\Longrightarrow$ *)
  $\exists(q_2,q_2') \in Q_r : \mathit{fsclean}[(q_1,q_1') \overset{\lambda_1}{\Longrightarrow}_r (q_2,q_2') \overset{\sigma_1}{\Longrightarrow}_r (q,q')]$
  $\Rightarrow$ (* Lemma B.2.14 *)
  $\exists(q_2,q_2') \in Q_r : \mathit{fsclean}[(q_1,q_1') \overset{\lambda_1}{\Longrightarrow}_r (q_2,q_2') \overset{\sigma_1}{\Longrightarrow}_r (q,q')]$
  $\wedge\, \exists\mu \in L : q_1 \xrightarrow{\mu} q_2 \wedge r(\mu) \overset{\lambda_1}{\Longrightarrow}_{r(\mu)} q_2'$

  In case $\sigma_2 = \epsilon$ we are done here ($q_2 = q$). Otherwise we continue as follows:

$\Rightarrow$ (* Lemma B.2.12, $q_2' \notin \mathsf{Final}$ because of the fsclean property *)
$\exists (q_2, q_2') \in Q_r : \mathit{fsclean}[(q_1, q_1') \xRightarrow{\lambda_1}_r (q_2, q_2') \xRightarrow{\sigma_1}_r (q, q')]$
$\wedge\, \exists \mu \in L : q_1 \xrightarrow{\mu} q_2 \wedge r(\mu) \xRightarrow{\lambda_1}_{r(\mu)} q_2' \wedge q_2 = q$
$\wedge\, \exists \mu' \in L_\tau : q_2' \xRightarrow{\sigma_1}_{r(\mu')} q'$

$\Rightarrow$ (* Logical reasoning *)
$\exists (q, q_2') \in Q_r : \mathit{fsclean}[(q_1, q_1') \xRightarrow{\lambda_1}_r (q, q_2') \xRightarrow{\sigma_1}_r (q, q')]$
$\wedge\, \exists \mu \in L : q_1 \xrightarrow{\mu} q \wedge r(\mu) \xRightarrow{\lambda_1}_{r(\mu)} q_2' \wedge \exists \mu' \in L_\tau : q_2' \xRightarrow{\sigma_1}_{r(\mu')} q'$

$\Rightarrow$ (* Definition 5.2.1 (LTS refinement): state sets of refinement transition systems are unique (so $\mu = \mu'$) *)
$\exists (q, q_2') \in Q_r : \mathit{fsclean}[(q_1, q_1') \xRightarrow{\lambda_1}_r (q, q_2') \xRightarrow{\sigma_1}_r (q, q')]$
$\wedge\, \exists \mu \in L : q_1 \xrightarrow{\mu} q \wedge r(\mu) \xRightarrow{\lambda_1}_{r(\mu)} q_2' \xRightarrow{\sigma_1}_{r(\mu)} q'$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\wedge\, \exists \mu \in L : q_1 \xrightarrow{\mu} q \wedge r(\mu) \xRightarrow{\lambda_1 \cdot \sigma_1}_{r(\mu)} q'$

- $\lambda_1 = \delta$. This case implies that $\sigma = \delta$ and that case is excluded from the lemma.

$\mathit{fsclean}[(q_0, q_0') \xRightarrow{\delta \cdot \sigma_1} (q, q')]$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xRightarrow{\delta} (q_1, q_1') \xRightarrow{\sigma_1} (q, q')]$
$\Rightarrow$ (* Lemma B.2.15 *)
$\exists (q_1, q_1') \in Q_r : \mathit{fsclean}[(q_0, q_0') \xRightarrow{\delta} (q_1, q_1') \xRightarrow{\sigma_1} (q, q')]$
$\wedge\, q_0 \xrightarrow{\delta} q_1 \wedge (q_0, q_0') = (q_1, q_1')$
$\Rightarrow$ (* fsclean property implies that $\sigma_1 = \epsilon$ and $(q_1, q_1') = (q, q')$ *)
$\mathit{fsclean}[(q_0, q_0') \xRightarrow{\delta} (q, q')]$

□

So far the lemmas implied properties of the abstract system, based on properties of the refined system. The following lemmas are the other way around. They are the building blocks for the proof of Proposition B.2.21 which expresses a relation between traces in the abstract system and the related traces in the refined system.

Lemma B.2.17 Let $q, q_1 \in Q$, where Q is the set of states of an arbitrary transition system.

$$q \xRightarrow{\epsilon} q_1 \Rightarrow \forall q' \in \mathsf{Final} : (\exists q_1' \in \mathsf{Final} : (q, q') \xRightarrow{\epsilon}_r (q_1, q_1'))$$

□

Proof We prove the following stronger statement. Let $n \geq 0$.

$$q \xrightarrow{\tau^n} q_1 \Rightarrow \forall q' \in \mathsf{Final} : (\exists q_1' \in \mathsf{Final} : (q, q') \xrightarrow{\tau^n}_r (q_1, q_1')) \qquad \text{(B.1)}$$

Proof by induction on n.

Basic step: $n = 0$.

$$\begin{array}{ll} & q \xrightarrow{\tau^0} q_1 \\ \Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\ & q \xrightarrow{\tau^0} q_1 \wedge q = q_1 \\ \Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\ & q = q_1 \wedge \forall q' \in \mathsf{Final} : (q, q') \xrightarrow{\tau^0}_r (q, q') \\ \Rightarrow & (*\ \text{Logical reasoning}\ *) \\ & \forall q' \in \mathsf{Final} : (\exists q_1' \in \mathsf{Final} : (q, q') \xrightarrow{\tau^0}_r (q_1, q_1')) \end{array}$$

Induction step: Assume that the lemma holds for $0 \leq j < n$. Let $\sigma = \tau^j \cdot \tau$.

$$\begin{array}{ll} & q \xrightarrow{\tau^j \cdot \tau} q_1 \\ \Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\ & \exists q_2 \in Q : q \xrightarrow{\tau^j} q_2 \xrightarrow{\tau} q_1 \\ \Rightarrow & (*\ \text{Induction}\ *) \\ & \exists q_2 \in Q : q \xrightarrow{\tau^j} q_2 \xrightarrow{\tau} q_1 \wedge \forall q' \in \mathsf{Final} : (\exists q_2' \in \mathsf{Final} : \\ & (q, q') \xrightarrow{\tau^j}_r (q_2, q_2')) \\ \Rightarrow & (*\ \text{Definition 5.2.1 (LTS refinement)}\ T_1\ *) \\ & \exists q_2 \in Q : q \xrightarrow{\tau^j} q_2 \xrightarrow{\tau} q_1 \wedge \forall q' \in \mathsf{Final} : (\exists q_2' \in \mathsf{Final} : \\ & (q, q') \xrightarrow{\tau^j}_r (q_2, q_2') \xrightarrow{\tau}_r (q_1, \mathsf{final}_\tau)) \\ \Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\ & \forall q' \in \mathsf{Final} : (q, q') \xrightarrow{\tau^j \cdot \tau}_r (q_1, \mathsf{final}_\tau) \\ \Rightarrow & (*\ \text{Logical reasoning}\ *) \\ & \forall q' \in \mathsf{Final} : (\exists q_1' \in \mathsf{Final} : (q, q') \xrightarrow{\tau^j \cdot \tau}_r (q_1, q_1') \end{array}$$

□

Lemma B.2.18 Let $q, q_1 \in Q, \lambda \in L_\delta$, where Q is the set of states of an arbitrary transition system.

$$q \overset{\lambda}{\Longrightarrow} q_1 \Rightarrow \forall q' \in \mathsf{Final}, \sigma \in \lambda[r]_{rc} : (\exists q_1 \in \mathsf{Final} : (q, q') \overset{\sigma}{\Longrightarrow}_r (q_1, q_1'))$$

□

Proof We identify the following cases:

- $\lambda \in L$.

$$\begin{array}{ll} & q \overset{\lambda}{\Longrightarrow} q_1 \\ \Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\ & \exists q_2, q_3 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\lambda} q_3 \overset{\epsilon}{\Longrightarrow} q_1 \\ \Rightarrow & (*\ \text{Lemma B.2.17}\ *) \\ & \exists q_2, q_3 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\lambda} q_3 \overset{\epsilon}{\Longrightarrow} q_1 \wedge \forall q', q_3' \in \mathsf{Final} : \\ & (\exists q_2', q_1' \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q_2') \wedge (q_3, q_3') \overset{\epsilon}{\Longrightarrow}_r (q_1, q_1')) \end{array}$$

$\Rightarrow$ (∗ Lemma B.2.6 ($\lambda \neq \delta$) ∗)
$\exists q_2, q_3 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\lambda} q_3 \overset{\epsilon}{\Longrightarrow} q_1 \wedge \forall q', q'_3 \in \mathsf{Final} :$
$(\exists q'_2, q'_1 \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2) \wedge (q_3, q'_3) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1))$
$\wedge \forall q'_4 \in \mathsf{Final}, \sigma \in \lambda[r]_{rc} : (q_2, q'_4) \overset{\sigma}{\Longrightarrow}_r (q_3, \mathsf{final}_{r(\lambda)})$
$\Rightarrow$ (∗ Logical reasoning ∗)
$\forall q' \in \mathsf{Final}, \sigma \in \lambda[r]_{rc} :$
$(\exists q'_1, q'_2 \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2) \overset{\sigma}{\Longrightarrow}_r (q_3, \mathsf{final}_{r(\lambda)}) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1))$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\forall q' \in \mathsf{Final}, \sigma \in \lambda[r]_{rc} : (\exists q'_1 \in \mathsf{Final} : (q, q') \overset{\sigma}{\Longrightarrow}_r (q_1, q'_1))$

- $\lambda = \delta$.

$q \overset{\delta}{\Longrightarrow} q_1$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists q_2, q_3 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\delta} q_3 \overset{\epsilon}{\Longrightarrow} q_1$
$\Rightarrow$ (∗ Definition δ ∗)
$\exists q_2 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\delta} q_2 \overset{\epsilon}{\Longrightarrow} q_1 \wedge \forall \mu \in U_\tau : q_2 \overset{\mu}{\not\longrightarrow}$
$\Rightarrow$ (∗ Lemma B.2.17 ∗)
$\exists q_2 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\delta} q_2 \overset{\epsilon}{\Longrightarrow} q_1 \wedge \forall \mu \in U_\tau : q_2 \overset{\mu}{\not\longrightarrow}$
$\wedge \forall q', q'_3 \in \mathsf{Final} : (\exists q'_1, q'_2 \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2)$
$\wedge (q_2, q'_3) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1))$
$\Rightarrow$ (∗ Definition 4.6.1 (Constraints on the refinement function) ∗)
$\exists q_2 \in Q : q \overset{\epsilon}{\Longrightarrow} q_2 \xrightarrow{\delta} q_2 \overset{\epsilon}{\Longrightarrow} q_1 \wedge \forall \mu \in U_\tau : q_2 \overset{\mu}{\not\longrightarrow}$
$\wedge \forall q', q'_3 \in \mathsf{Final} : (\exists q'_1, q'_2 \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2)$
$\wedge (q_2, q'_3) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1))$
$\Rightarrow$ (∗ Definition 5.2.1 (LTS refinement T_1)
and Definition 4.6.1 (constraints 1, 2, 3) ∗)
$\forall q', q'_3 \in \mathsf{Final} : (\exists q'_1, q'_2 \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2)$
$\wedge (q_2, q'_3) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1)) \wedge \forall q'_4 \in \mathsf{Final}, \mu' \in U_{r\tau} : (q_2, q'_4) \overset{\mu'}{\not\longrightarrow}_r$
$\Rightarrow$ (∗ Definition δ ∗)
$\forall q', q'_3 \in \mathsf{Final} : (\exists q'_1, q'_2 \in \mathsf{Final} : (q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2)$
$\wedge (q_2, q'_3) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1)) \wedge \forall q'_4 \in \mathsf{Final} : (q_2, q'_4) \xrightarrow{\delta}_r (q_2, q'_4)$
$\Rightarrow$ (∗ Logical reasoning ∗)
$\forall q' \in \mathsf{Final} : (\exists q'_1, q'_2 \in \mathsf{Final} :$
$(q, q') \overset{\epsilon}{\Longrightarrow}_r (q_2, q'_2) \xrightarrow{\delta}_r (q_2, q'_2) \overset{\epsilon}{\Longrightarrow}_r (q_1, q'_1))$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\forall q' \in \mathsf{Final} : (\exists q'_1 \in \mathsf{Final} : (q, q') \overset{\delta}{\Longrightarrow}_r (q_1, q'_1))$
$\Rightarrow$ (∗ Definition 5.3.3 ∗)
$\forall q' \in \mathsf{Final}, \sigma \in \delta[r]_{rc} : (\exists q'_1 \in \mathsf{Final} : (q, q') \overset{\sigma}{\Longrightarrow}_r (q_1, q'_1))$

□

Lemma B.2.19 Let $q, q_1 \in Q, \sigma \in L^*_\delta$, where Q is the set of states of an arbitrary transition system.

$$q \overset{\sigma}{\Longrightarrow} q_1 \Rightarrow \forall q' \in \mathsf{Final}, \sigma' \in \sigma[r]_{rc} : (\exists q'_1 \in \mathsf{Final} : (q, q') \overset{\sigma'}{\Longrightarrow}_r (q_1, q'_1))$$

□

Proof Proof by induction on the length of σ.

Basic step: $\sigma = \epsilon$. This step follows from Lemma B.2.17 ($\epsilon[r] = \{\epsilon\}$).

Induction step: Let $\sigma = \sigma_1 \cdot \lambda$ with $\sigma_1 \in L_\delta^*$ and $\lambda \in L_\delta$ and assume that the lemma holds for σ_1.

$$
\begin{array}{ll}
 & q \xRightarrow{\sigma_1 \cdot \lambda} q_1 \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists q_2 \in Q : q \xRightarrow{\sigma_1} q_2 \xRightarrow{\lambda} q_1 \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists q_2 \in Q : q \xRightarrow{\sigma_1} q_2 \xRightarrow{\lambda} q_1 \wedge \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc} : \\
 & (\exists q_2' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma_1'}_r (q_2, q_2')) \\
\Rightarrow & (*\ \text{Lemma B.2.18}\ *) \\
 & \exists q_2 \in Q : q \xRightarrow{\sigma_1} q_2 \xRightarrow{\lambda} q_1 \wedge \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc} : \\
 & (\exists q_2' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma_1'}_r (q_2, q_2')) \\
 & \forall q_3' \in \mathsf{Final}, \sigma_2' \in \lambda[r]_{rc} : (\exists q_1' \in \mathsf{Final} : (q_2, q_3') \xRightarrow{\sigma_2'}_r (q_1, q_1')) \\
\Rightarrow & (*\ \text{Logical reasoning}\ *) \\
 & \exists q_2 \in Q : q \xRightarrow{\sigma_1} q_2 \xRightarrow{\lambda} q_1 \wedge \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \lambda[r]_{rc} : \\
 & (\exists q_1' q_2' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma_1'}_r (q_2, q_2') \xRightarrow{\sigma_2'}_r (q_1, q_1')) \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \lambda[r]_{rc} : \\
 & (\exists q_1' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma_1' \cdot \sigma_2'}_r (q_1, q_1')) \\
\Rightarrow & (*\ \text{Lemma B.1.3}\ *) \\
 & \forall q' \in \mathsf{Final}, \sigma' \in (\sigma_1 \cdot \lambda)[r]_{rc} : (\exists q_1' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma'}_r (q_1, q_1')) \\
\Rightarrow & (*\ \text{Premise: } \sigma = \sigma_1 \cdot \lambda\ \ *) \\
 & \forall q' \in \mathsf{Final}, \sigma' \in \sigma[r]_{rc} : (\exists q_1' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma'}_r (q_1, q_1'))
\end{array}
$$

□

Lemma B.2.20 Let $q \in Q, \lambda \in L$, where Q is the set of states of an arbitrary transition system.

$$q \xRightarrow{\lambda} \ \Rightarrow \forall q' \in \mathsf{Final}, \sigma \in \lambda[r]_{inc} : (q, q') \xRightarrow{\sigma}_r$$

□

Proof

$$\begin{array}{ll}
 & q \overset{\lambda}{\Longrightarrow} q_1 \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists q_1 \in Q : q \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{\lambda} \\
\Rightarrow & (*\ \text{Lemma B.2.17}\ *) \\
 & \exists q_1 \in Q : q \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{\lambda} \wedge \forall q' \in \mathsf{Final} : (\exists q_1' \in \mathsf{Final} : (q,q') \overset{\epsilon}{\Longrightarrow}_r (q_1, q_1')) \\
\Rightarrow & (*\ \text{Lemma B.2.7}\ *) \\
 & \exists q_1 \in Q : q \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{\lambda} \wedge \forall q' \in \mathsf{Final} : (\exists q_1' \in \mathsf{Final} : (q,q') \overset{\epsilon}{\Longrightarrow}_r (q_1, q_1')) \\
 & \wedge \forall q_2' \in \mathsf{Final}, \sigma \in \lambda[r]_{inc} : (q_1, q_2') \overset{\sigma}{\Longrightarrow}_r \\
\Rightarrow & (*\ \text{Logical reasoning}\ *) \\
 & \forall q' \in \mathsf{Final}, \sigma \in \lambda[r]_{inc} : (\exists q_1' \in \mathsf{Final} : (q,q') \overset{\epsilon}{\Longrightarrow}_r (q_1, q_1') \overset{\sigma}{\Longrightarrow}_r) \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \forall q' \in \mathsf{Final}, \sigma \in \lambda[r]_{inc} : (q,q') \overset{\sigma}{\Longrightarrow}_r
\end{array}$$

□

Proposition B.2.21 Let $\sigma \in L_\delta^*$

$$q \overset{\sigma}{\Longrightarrow} \ \Rightarrow \forall q' \in \mathsf{Final}, \sigma' \in \sigma[r] : (q, q') \overset{\sigma'}{\Longrightarrow}_r$$

□

Proof For completely refined traces ($\sigma' \in \sigma[r]_{rc}$) this lemma follows from Lemma B.2.19. The remainder of the proof is for $\sigma' \in \sigma[r]_{inc}$.

Proof by induction on the length of σ.

Only if: The case for ϵ is vacuously true because $\epsilon[r]_{inc} = \emptyset$.

If: Suppose $\sigma = \sigma_1 \cdot \lambda$ with $\sigma_1 \in L_\delta^*$ and $\lambda \in L_\delta$.

$$\begin{array}{ll}
 & q \overset{\sigma_1 \cdot \lambda}{\Longrightarrow} \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists q_1 \in Q : q \overset{\sigma_1}{\Longrightarrow} q_1 \overset{\lambda}{\Longrightarrow} \\
\Rightarrow & (*\ \text{Lemma B.2.19}\ *) \\
 & \exists q_1 \in Q : q \overset{\sigma_1}{\Longrightarrow} q_1 \overset{\lambda}{\Longrightarrow} \\
 & \wedge \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc} : (\exists q_1' \in \mathsf{Final} : (q,q') \overset{\sigma_1'}{\Longrightarrow}_r (q_1, q_1')) \\
\Rightarrow & (*\ \text{Lemma B.2.20}\ *) \\
 & \exists q_1 \in Q : q \overset{\sigma_1}{\Longrightarrow} q_1 \overset{\lambda}{\Longrightarrow} \\
 & \wedge \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc} : (\exists q_1' \in \mathsf{Final} : (q,q') \overset{\sigma_1'}{\Longrightarrow}_r (q_1, q_1')) \\
 & \wedge \forall q_3' \in \mathsf{Final}, \sigma_2' \in \lambda[r]_{inc} : (q_1, q_3') \overset{\sigma_2'}{\Longrightarrow}_r
\end{array}$$

In case $\lambda = \delta$, we have $\delta[r]_{inc} = \emptyset$ and we are done. Otherwise we continue as follows:

$\Rightarrow$ (* Logical reasoning *)
$\exists q_1 \in Q : q \xRightarrow{\sigma_1} q_1 \xRightarrow{\lambda} \wedge \forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \lambda[r]_{inc} :$
$(\exists q_1' \in \mathsf{Final} : (q, q') \xRightarrow{\sigma_1'}_r (q_1, q_1') \xRightarrow{\sigma_2'}_r)$
$\Rightarrow$ (* Definition $\Longrightarrow$ *)
$\forall q' \in \mathsf{Final}, \sigma_1' \in \sigma_1[r]_{rc}, \sigma_2' \in \lambda[r]_{inc} : (q, q') \xRightarrow{\sigma_1' \cdot \sigma_2'}_r$
$\Rightarrow$ (* Lemma B.1.4 *)
$\forall q' \in \mathsf{Final}, \sigma' \in (\sigma_1 \cdot \lambda)[r] : (q, q') \xRightarrow{\sigma'}_r$
$\Rightarrow$ (* Premise: $\sigma = \sigma_1 \cdot \lambda$ *)
$\forall q' \in \mathsf{Final}, \sigma' \in \sigma[r] : (q, q') \xRightarrow{\sigma'}_r$

□

Theorem 5.3.12 Let $s \in \mathbf{LTS}(I, U), L = I \cup U, r : L_\tau \to \mathbf{FLTS}(I', U')$ with $I', U' \subseteq \mathbf{L}$.

$$\mathit{Straces}(s)[r] = \mathit{Straces}(s[r])$$

□

Proof

Only if: Let $\sigma' \in \mathit{Straces}(s)$ such that $\sigma \in \sigma'[r]$

$\Rightarrow$ (* Definition *Straces* *)
$\mathsf{start}_s \xRightarrow{\sigma'}$
$\Rightarrow$ (* Proposition B.2.21 *)
$\forall q' \in \mathsf{Final}, \sigma'' \in \sigma'[r] : (\mathsf{start}_s, q') \xRightarrow{\sigma''}_r$
$\Rightarrow$ (* Premise: $\sigma \in \sigma'[r]$ *)
$\forall q' \in \mathsf{Final} : (\mathsf{start}_s, q') \xRightarrow{\sigma}_r$
$\Rightarrow$ (* $\checkmark \in \mathsf{Final}$ *)
$(\mathsf{start}_s, \checkmark) \xRightarrow{\sigma}_r$
$\Rightarrow$ (* $(\mathsf{start}_s, \checkmark) = \mathsf{start}_{s[r]}$ *)
$s[r] \xRightarrow{\sigma}_r$
$\Rightarrow$ (* Definition *Straces* *)
$\sigma \in \mathit{Straces}(s[r])$

If: $\sigma \in \mathit{Straces}(s[r])$. We first split the trace σ into sub traces, in such a way that every sub trace ends in a state pair where the second element is a final state of a refinement transition system. We split σ exactly at the places where it encounters a state with a final state as the second element of its state pair. For the record, we do not split on empty traces where the start and end state of the transition are the same, i.e., empty traces that consist of zero τ steps. This means that in cases with an empty trace only the second part of the or-clause of the implication of Lemma B.2.13 holds.

$$
\begin{array}{ll}
 & \sigma \in \mathit{Straces}(s[r]) \\
\Rightarrow & (*\ \text{Definition } \mathit{Straces}\ \ *) \\
 & \exists (q,q') \in Q_r : s[r] \overset{\sigma}{\Longrightarrow}_r (q,q') \\
\Rightarrow & (*\ \text{Definition } \Longrightarrow \text{ and Definition 5.2.1}\ \ *) \\
 & \exists n \geq 0, \sigma_1,\ldots,\sigma_n \in L^*_{r\delta}, q_1,\ldots,q_{n-1} \in Q, q'_1,\ldots,q'_{n-1} \in \mathsf{Final} : \\
 & \sigma = \sigma_1 \cdots \sigma_n \wedge s[r] \overset{\sigma_1}{\Longrightarrow}_r (q_1,q'_1) \cdots (q_{n-1},q_{n-1}) \overset{\sigma_n}{\Longrightarrow}_r (q,q') \\
 & \wedge \forall 1 \leq i < n-1 : \mathit{fsclean}[(q_i,q'_i) \overset{\sigma_i}{\Longrightarrow}_r (q_{i+1},q'_{i+1})] \\
 & \wedge\, \mathit{fsclean}[s[r] \overset{\sigma_1}{\Longrightarrow}_r (q_1,q'_1)] \wedge \mathit{fsclean}[(q_{n-1},q'_{n-1}) \overset{\sigma_n}{\Longrightarrow}_r (q,q')]
\end{array}
$$

To make the proof easier to read we rename q to q_n and q' to q'_n and we use (q_0, q'_0) as the start state of $s[r]$. We identify the following cases:

$\sigma = \epsilon$

$$
\begin{array}{ll}
 & q_0 \overset{\epsilon}{\Longrightarrow} \\
\Rightarrow & (*\ \text{Definition Straces, premise: } q_0 \text{ is start state of } s\ \ *) \\
 & \epsilon \in \mathit{Straces}(s) \\
\Rightarrow & (*\ \text{Definition 5.3.3 (complete trace refinement)}\ \ *) \\
 & \epsilon \in \mathit{Straces}(s)[r]
\end{array}
$$

$\sigma = \delta.$

The δ action can have zero or more ϵ steps before and after it. Suppose that $\sigma_j = \delta$ for some $1 \leq j \leq n$.

$$
\begin{array}{ll}
 & \exists n \geq 0, \sigma_1,\ldots,\sigma_n \in L^*_{r\delta}, q_1,\ldots,q_{n-1} \in Q, q'_1,\ldots,q'_{n-1} \in \mathsf{Final} : \\
 & \sigma = \sigma_1 \cdots \sigma_n \wedge (q_0,q'_0) \overset{\sigma_1}{\Longrightarrow}_r (q_1,q'_1) \cdots (q_{n-1},q'_{n-1}) \overset{\sigma_n}{\Longrightarrow}_r (q_n,q'_n) \\
 & \wedge \forall 0 \leq i < n : \mathit{fsclean}[(q_i,q'_i) \overset{\sigma_{i+1}}{\Longrightarrow}_r (q_{i+1},q_{i+1})] \\
 & \wedge \exists 1 \leq j \leq n : \sigma_j = \delta \\
\Rightarrow & (*\ \text{Lemma B.2.13}\ \ *) \\
 & \exists n \geq 0, \sigma_1,\ldots,\sigma_n \in L^*_{r\delta}, q_1,\ldots,q_{n-1} \in Q, q'_1,\ldots,q'_{n-1} \in \mathsf{Final} : \\
 & \sigma = \sigma_1 \cdots \sigma_n \wedge (q_0,q'_0) \overset{\sigma_1}{\Longrightarrow}_r (q_1,q'_1) \cdots (q_{n-1},q'_{n-1}) \overset{\sigma_n}{\Longrightarrow}_r (q_n,q'_n) \\
 & \wedge \forall 0 \leq i < n : \mathit{fsclean}[(q_i,q'_i) \overset{\sigma_{i+1}}{\Longrightarrow}_r (q_{i+1},q_{i+1})] \\
 & \wedge \exists 1 \leq j \leq n : \sigma_j = \delta \wedge q_0 \xrightarrow{\tau^j} q_j \wedge q_{j+1} \xrightarrow{\tau^{n-j-1}} q_n \\
\Rightarrow & (*\ \text{Lemma B.2.15}\ \ *) \\
 & \exists n \geq 0, q_1,\ldots,q_{n-1} \in Q : \\
 & \wedge \exists 1 \leq j \leq n : \sigma_j = \delta \wedge q_0 \xrightarrow{\tau^j} q_j \wedge q_{j+1} \xrightarrow{\tau^{n-j-1}} q_n \wedge q_j \xrightarrow{\delta} q_{j+1} \\
\Rightarrow & (*\ \text{Definition } \Longrightarrow\ \ *) \\
 & q_0 \overset{\delta}{\Longrightarrow} \\
\Rightarrow & (*\ \text{Definition } \mathit{Straces}\ \ *) \\
 & \delta \in \mathit{Straces}(s) \\
\Rightarrow & (*\ \text{Definition 5.3.5}\ \ *) \\
 & \delta \in \mathit{Straces}(s)[r]
\end{array}
$$

$\sigma \notin \{\epsilon, \delta\}$

$$
\begin{array}{ll}
 & \exists n \geq 0, \sigma_1, \ldots, \sigma_n \in L_{r\delta}^*, q_1, \ldots, q_{n-1} \in Q, q_1', \ldots, q_{n-1}' \in \mathsf{Final} : \\
 & \sigma = \sigma_1 \cdots \sigma_n \wedge (q_0, q_0') \xrightarrow{\sigma_1}_r (q_1, q_1') \cdots (q_{n-1}, q_{n-1}') \xrightarrow{\sigma_n}_r (q_n, q_n') \\
 & \wedge \forall 0 \leq i < n : \mathit{fsclean}[(q_i, q_i') \xrightarrow{\sigma_{i+1}}_r (q_{i+1}, q_{i+1})] \\
\Rightarrow & (*\ \text{Lemma B.2.16 and Lemma B.2.15}\ *) \\
 & \exists n \geq 0, \sigma_1, \ldots \sigma_n \in L_{r\delta}^*, q_1, \ldots, q_{n-1} \in Q, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : \\
 & \sigma = \sigma_1 \cdots \sigma_n \wedge q_0 \xrightarrow{\lambda_1} q_1 \cdots q_{n-1} \xrightarrow{\lambda_n} q_n \\
 & \wedge \forall 1 \leq i \leq n : r(\lambda_i) \overset{\sigma_i}{\Longrightarrow} q_i' \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists n \geq 0, \sigma_1, \ldots \sigma_n \in L_{r\delta}^*, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : \sigma = \sigma_1 \cdots \sigma_n \\
 & \wedge q_0 \xrightarrow{\lambda_1 \cdots \lambda_n} q_n \wedge \forall 1 \leq i \leq n : r(\lambda_i) \overset{\sigma_i}{\Longrightarrow} q_i' \\
\Rightarrow & (*\ \text{Logical reasoning: } \lambda_i \text{ may be } \tau,\ q_0 \text{ is the start state of } s, \\
 & \quad \text{we use Definition 2.3.5 for trace projection.}\ *) \\
 & \exists n \geq 0, \sigma_1, \ldots, \sigma_n \in L_{r\delta}^*, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : \sigma = \sigma_1 \cdots \sigma_n \\
 & \wedge \forall 1 \leq i \leq n : r(\lambda_i) \overset{\sigma_i}{\Longrightarrow} q_i' \wedge s \overset{(\lambda_1 \cdots \lambda_n) \upharpoonright L_\delta}{\Longrightarrow}
\end{array}
$$

At this point we identify the following cases.

- $q_n' \in \mathsf{Final}$

$$
\begin{array}{ll}
\Rightarrow & (*\ \text{Definition 5.3.1 } (\mathit{TXStraces})\ *) \\
 & \exists n \geq 0, \sigma_1, \ldots, \sigma_n \in L_{r\delta}^*, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : \sigma = \sigma_1 \cdots \sigma_n \\
 & \wedge s \overset{\lambda_1 \cdots \lambda_n) \upharpoonright L_\delta}{\Longrightarrow} \wedge \forall 1 \leq i \leq n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \\
\Rightarrow & (*\ \text{Definition 5.3.3 (complete trace refinement)}\ *) \\
 & \exists n \geq 0, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : s \overset{(\lambda_1 \cdots \lambda_n) \upharpoonright L_\delta}{\Longrightarrow} \\
 & \wedge \sigma \in ((\lambda_1 \cdots \lambda_n) \upharpoonright L_\delta)[r] \\
\Rightarrow & (*\ \text{Logical reasoning using Definition 2.3.5 (projection)}\ *) \\
 & \exists \sigma' \in L_\delta^* : s \overset{\sigma'}{\Longrightarrow} \wedge \sigma \in \sigma'[r] \\
\Rightarrow & (*\ \text{Definition } \mathit{Straces}\ *) \\
 & \exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r] \\
\Rightarrow & (*\ \text{Logical reasoning}\ *) \\
 & \sigma \in \mathit{Straces}(s)[r]
\end{array}
$$

- $q_n' \notin \mathsf{Final}$

$$
\begin{array}{ll}
\Rightarrow & (*\ \text{Definition 5.3.2 } (\mathit{XStraces})\ *) \\
 & \exists n \geq 0, \sigma_1, \ldots, \sigma_n \in L_{r\delta}^*, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : \sigma = \sigma_1 \cdots \sigma_n \\
 & \wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \\
 & \wedge \sigma_n \in \mathit{XStraces}(r(\lambda_n)) \\
\Rightarrow & (*\ \text{Definition 5.3.4 (incomplete trace refinement)}\ *) \\
 & \exists n \geq 0, \lambda_1, \ldots, \lambda_n \in L_{\delta\tau} : s \overset{(\lambda_1 \cdots \lambda_n) \upharpoonright L_\delta}{\Longrightarrow} \\
 & \wedge \sigma \in ((\lambda_1 \cdots \lambda_n) \upharpoonright L_\delta)[r]
\end{array}
$$

$\Rightarrow$ (∗ Logical reasoning using Definition 2.3.5 (projection) ∗)
$\exists \sigma' \in L_\delta^* : s \overset{\sigma'}{\Longrightarrow} \wedge \sigma \in \sigma'[r]$
$\Rightarrow$ (∗ Definition *Straces* ∗)
$\exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r]$
$\Rightarrow$ (∗ Logical reasoning ∗)
$\sigma \in \mathit{Straces}(s)[r]$

□

B.3 Proofs Section 5.4: ioco with refinement

Proposition 5.4.6

$$\mathit{out}(s[r] \textbf{ after } \sigma) = \mathit{out}_{rc}(s, \sigma, r) \cup \mathit{out}_{inc}(s, \sigma, r)$$

□

Proof

$\supseteq$ We prove that $x \in \mathit{out}_{rc}(s, \sigma, r) \cup \mathit{out}_{inc}(s, \sigma, r) \Rightarrow x \in \mathit{out}(s[r] \textbf{ after } \sigma)$. We identify the following cases:

$x \in \mathit{out}_{rc}(s, \sigma, r)$ and $x \neq \delta$. Because $x \neq \delta$ we can drop the part about δ in Definition 5.4.1: $\bigcup_{\sigma' \in \Sigma} \mathit{out}(s \textbf{ after } \sigma') \cap \{\delta\}$.

$\Rightarrow$ (∗ Definition 5.4.1 (out_{rc}) ∗)
$\exists \sigma' \in \sigma\langle r\rangle_{rc}, \mu \in \mathit{out}(s \textbf{ after } \sigma') \backslash \{\delta\} : \sigma' \in \mathit{Straces}(s)$
$\wedge\, x \in \mathit{out}(r(\mu) \textbf{ after } \epsilon) \backslash \{\delta\}$
$\Rightarrow$ (∗ Definition *Straces* ∗)
$\exists \sigma' \in \sigma\langle r\rangle_{rc}, \mu \in U : x \in \mathit{out}(r(\mu) \textbf{ after } \epsilon) \backslash \{\delta\}$
$\wedge\, \sigma' \cdot \mu \in \mathit{Straces}(s)$
$\Rightarrow$ (∗ Definition 5.3.5 (trace refinement) ∗)
$\exists \sigma' \in \sigma\langle r\rangle_{rc}, \mu \in U : x \in \mu[r] \wedge \sigma' \cdot \mu \in \mathit{Straces}(s)$
$\Rightarrow$ (∗ Lemma B.1.1 ∗)
$\exists \sigma' \in \sigma\langle r\rangle_{rc}, \mu \in U : x \in \mu[r]$
$\wedge\, \sigma' \cdot \mu \in \mathit{Straces}(s) \wedge \sigma \in \sigma'[r]_{rc}$
$\Rightarrow$ (∗ Lemma B.1.5 ∗)
$\exists \sigma' \in \sigma\langle r\rangle_{rc}, \mu \in U : \sigma' \cdot \mu \in \mathit{Straces}(s) \wedge \sigma \cdot x \in (\sigma' \cdot \mu)[r]$
$\Rightarrow$ (∗ Set operations ∗)
$\sigma \cdot x \in \mathit{Straces}(s)[r]$
$\Rightarrow$ (∗ Theorem 5.3.12 ∗)
$\sigma \cdot x \in \mathit{Straces}(s[r])$
$\Rightarrow$ (∗ Definitions *out* and **after** ∗)
$x \in \mathit{out}(s[r] \textbf{ after } \sigma)$

$x \in out_{rc}(s, \sigma, r)$ and $x = \delta$

$\Rightarrow$ (* Definition 5.4.1 (out_{rc}) *)
$\exists \sigma' \in \sigma\langle r\rangle_{rc} : \sigma' \in Straces(s) \wedge \delta \in out(s \textbf{ after } \sigma')$
$\Rightarrow$ (* Definition *Straces* *)
$\exists \sigma' \in \sigma\langle r\rangle_{rc} : \sigma'\cdot\delta \in Straces(s)$
$\Rightarrow$ (* Lemma B.1.2 *)
$\exists \sigma' \in \sigma\langle r\rangle_{rc} : \sigma'\cdot\delta \in Straces(s) \wedge \sigma \in \sigma'[r]_{rc}$
$\Rightarrow$ (* Definition 5.3.3 (trace refinement) *)
$\exists \sigma' \in \sigma\langle r\rangle_{rc} : \sigma'\cdot\delta \in Straces(s) \wedge \sigma\cdot\delta \in (\sigma'\cdot\delta)[r]_{rc}$
$\Rightarrow$ (* Set operations *)
$\sigma\cdot\delta \in Straces(s)[r]$
$\Rightarrow$ (* Theorem 5.3.12 *)
$\sigma\cdot\delta \in Straces(s[r])$
$\Rightarrow$ (* Definition *out* and **after** *)
$\delta \in out(s[r] \textbf{ after } \sigma)$

$x \in out_{inc}(s, \sigma, r)$

$\Rightarrow$ (* Definition 5.4.3 (out_{inc}) *)
$\exists \sigma_1 \cdots \sigma_n \in L_{r\delta^*}, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge (\lambda_1 \cdots \lambda_n) \in Straces(s) \wedge \forall 1 \leq i < n : \sigma_i \in TXStraces(r(\lambda_i))$
$\wedge \sigma_n \in XStraces(r(\lambda_n)) \wedge x \in out(r(\lambda_n) \textbf{ after } \sigma_n)$
$\Rightarrow$ (* Definition 5.3.3 (complete trace refinement) *)
$\exists \sigma_1 \cdots \sigma_n \in L_{r\delta}^*, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge (\lambda_1 \cdots \lambda_n) \in Straces(s) \wedge \sigma_1 \cdots \sigma_{n-1} \in (\lambda_1 \cdots \lambda_{n-1})[r]_{rc}$
$\wedge \sigma_n \in XStraces(r(\lambda_n)) \wedge x \in out(r(\lambda_n) \textbf{ after } \sigma_n)$
$\Rightarrow$ (* Definition 5.3.1 (*TXStraces*), Definition 5.3.2 (*XStraces*) *)
$\exists \sigma_1 \cdots \sigma_n \in L_{r\delta}^*, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge (\lambda_1 \cdots \lambda_n) \in Straces(s) \wedge \sigma_1 \cdots \sigma_{n-1} \in (\lambda_1 \cdots \lambda_{n-1})[r]_{rc}$
$\wedge \sigma_n\cdot x \in (XStraces(r(\lambda_n)) \cup TXStraces(r(\lambda_n)))$
$\Rightarrow$ (* Definition 5.3.3, Definition 5.3.4, Definition 5.3.5
complete, incomplete and general trace refinement) *)
$\exists \sigma_1 \cdots \sigma_n \in L_{r\delta}^*, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge (\lambda_1 \cdots \lambda_n) \in Straces(s) \wedge \sigma_1 \cdots \sigma_{n-1} \in (\lambda_1 \cdots \lambda_{n-1})[r]_{rc}$
$\wedge \sigma_n\cdot x \in \lambda_n[r]$
$\Rightarrow$ (* Lemma B.1.5 *)
$\exists \sigma_1 \cdots \sigma_n \in L_{r\delta}^*, \lambda_1 \cdots \lambda_n \in L_\delta : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge (\lambda_1 \cdots \lambda_n) \in Straces(s) \wedge (\sigma_1 \cdots \sigma_n\cdot x) \in (\lambda_1 \cdots \lambda_n)[r]$
$\Rightarrow$ (* Logical reasoning *)
$\exists (\lambda_1 \cdots \lambda_n) \in Straces(s) : \sigma\cdot x \in (\lambda_1 \cdots \lambda_n)[r]$
$\Rightarrow$ (* Set operations *)
$\sigma\cdot x \in Straces(s)[r]$

$\Rightarrow$ (* Theorem 5.3.12 *)
$\sigma{\cdot}x \in \mathit{Straces}(s[r])$
$\Rightarrow$ (* Definition *out* and **after** *)
$x \in \mathit{out}(s[r]\ \textbf{after}\ \sigma)$

$\subseteq$ We prove that $x \in \mathit{out}(s[r]\ \textbf{after}\ \sigma) \Rightarrow x \in \mathit{out}_{rc}(s,\sigma,r) \cup \mathit{out}_{inc}(s,\sigma,r)$.

$x \in \mathit{out}(s[r]\ \textbf{after}\ \sigma)$
$\Rightarrow$ (* Definition *out* and **after** *)
$\sigma{\cdot}x \in \mathit{Straces}(s[r]) \wedge x \in U_{r\delta}$
$\Rightarrow$ (* Theorem 5.3.12 *)
$\sigma{\cdot}x \in \mathit{Straces}(s)[r] \wedge x \in U_{r\delta}$
$\Rightarrow$ (* Logical reasoning *)
$\exists \sigma' \in \mathit{Straces}(s) : \sigma{\cdot}x \in \sigma'[r] \wedge x \in U_{r\delta}$

To keep the proof concise, we assume that $\sigma' = \lambda_1 \cdots \lambda_n$ for some $n \geq 0$ and $\forall 1 \leq i \leq n : \lambda_i \in L_\delta$, so $\lambda_1 \cdots \lambda_n \in \mathit{Straces}(s)$

$\Rightarrow$ (* Definition 5.3.3 (complete trace refinement)
and Definition 5.3.4 (incomplete trace refinement) *)
$\exists \sigma_1, \ldots, \sigma_n \in L^*_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge (\sigma_n{\cdot}x \in \mathit{TXStraces}(r(\lambda_n)) \vee \sigma_n{\cdot}x \in \mathit{XStraces}(r(\lambda_n)))$

We identify the following cases.
- σ is completely refined.

$\Rightarrow$ (* Definition 5.3.3 (complete trace refinement) *)
$\exists \sigma_1, \ldots, \sigma_n \in L^*_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge (\sigma_n{\cdot}x \in \mathit{TXStraces}(r(\lambda_n)) \vee \sigma_n{\cdot}x \in \mathit{XStraces}(r(\lambda_n))) \wedge \sigma_n = \epsilon$
$\Rightarrow$ (* Logical reasoning *)
$\exists \sigma_1, \ldots, \sigma_{n-1} \in L^*_{r\delta} : \sigma = \sigma_1 \cdots \sigma_{n-1}$
$\wedge \forall 1 \leq i \leq n-1 : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge (x \in \mathit{TXStraces}(r(\lambda_n)) \vee x \in \mathit{XStraces}(r(\lambda_n)))$

We identify the following cases:

- $x \in U_r$

  $\Rightarrow$ (* Definitions *out* and **after** with logical reasoning *)
  $\exists \sigma_1, \ldots, \sigma_{n-1} \in L^*_{r\delta} : \sigma = \sigma_1 \cdots \sigma_{n-1} \wedge \forall 1 \leq i \leq n-1 :$
  $\sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \wedge (x \in \mathit{TXStraces}(r(\lambda_n))$
  $\vee\, x \in \mathit{XStraces}(r(\lambda_n))) \wedge x \in \mathit{out}(r(\lambda_n)\ \textbf{after}\ \epsilon) \backslash \{\delta\}$
  $\Rightarrow$ (* Definition 4.6.1 1,2,3 (only refinements of an output
  action start with outputs) *)
  $\exists \sigma_1, \ldots, \sigma_{n-1} \in L^*_{r\delta} : \sigma = \sigma_1 \cdots \sigma_{n-1}$
  $\wedge \forall 1 \leq i \leq n-1 : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
  $\wedge\, x \in \mathit{out}(r(\lambda_n)\ \textbf{after}\ \epsilon) \backslash \{\delta\} \wedge \lambda_n \in U$

Note that the case $\lambda_n = \delta$ is ruled out by the premise that $x \in U_r$. In case $\lambda_n = \delta$ we get $x \in \mathit{TXStraces}(r(\delta)) \vee x \in \mathit{XStraces}(r(\delta))$, which implies $x \in \{\delta\}$.

$\Rightarrow$ (∗ Definitions *out* and **after** with $\lambda_1 \cdots \lambda_n \in \mathit{Straces}(s)$ ∗)
$\exists \sigma_1, \ldots, \sigma_{n-1} \in L_{r\delta}^* : \sigma = \sigma_1 \cdots \sigma_{n-1} \wedge \forall 1 \leq i \leq n-1 :$
$\sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \wedge x \in \mathit{out}(r(\lambda_n)\ \textbf{after}\ \epsilon) \backslash \{\delta\}$
$\wedge\, \lambda_n \in \mathit{out}(s\ \textbf{after}\ \lambda_1 \cdots \lambda_{n-1}) \backslash \{\delta\}$

$\Rightarrow$ (∗ Definition 5.3.7 (complete trace contraction) ∗)
$x \in \mathit{out}(r(\lambda_n)\ \textbf{after}\ \epsilon) \backslash \{\delta\}$
$\wedge\, \lambda_n \in \mathit{out}(s\ \textbf{after}\ \lambda_1 \cdots \lambda_{n-1}) \backslash \{\delta\} \wedge \lambda_1 \cdots \lambda_{n-1} \in \sigma \langle r \rangle_{rc}$

$\Rightarrow$ (∗ Definition 5.4.1 ∗)
$x \in \mathit{out}_{rc}(s, \sigma, r)$

- $x = \delta$. Because $\mathit{TXStraces}(\delta) = \{\delta\}$ and there is no other way to obtain δ as the first label of *XStraces* or *TXStraces*, we know that $\lambda_n = \delta$.

$\Rightarrow$ (∗ Definition 5.3.1 (*TXStraces*), Definition 5.3.2 (*XStraces*) and $\mathit{TXStraces}(r(\delta)) = \{\delta\}$ ∗)
$\exists \sigma_1, \ldots, \sigma_{n-1} \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_{n-1} \wedge \forall 1 \leq i \leq n-1 :$
$\sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \wedge x \in \mathit{TXStraces}(r(\lambda_n)) \wedge \lambda_n = \delta$

$\Rightarrow$ (∗ Definitions *out* and **after**, note that $\lambda_1 \cdots \lambda_n \in \mathit{Straces}(s)$ ∗)
$\exists \sigma_1, \ldots, \sigma_{n-1} \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_{n-1} \wedge \forall 1 \leq i \leq n-1 :$
$\sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \wedge x \in \mathit{TXStraces}(r(\lambda_n)) \wedge \lambda_n = \delta$
$\wedge\, \lambda_n \in \mathit{out}(s\ \textbf{after}\ \lambda_1 \cdots \lambda_{n-1})$

$\Rightarrow$ (∗ Definition 5.3.7 (complete trace contraction) ∗)
$\lambda_n = \delta \wedge \lambda_n \in \mathit{out}(s\ \textbf{after}\ \lambda_1 \cdots \lambda_{n-1})$
$\wedge\, \lambda_1 \cdots \lambda_{n-1} \in \sigma \langle r \rangle_{rc}$

$\Rightarrow$ (∗ Definition 5.4.1, note that $x = \delta$ ∗)
$x \in \mathit{out}_{rc}(s, \sigma, r)$

- σ is incompletely refined. In this case we know that $\sigma_n \neq \epsilon$ (using Definition 5.3.4 (incomplete trace refinement)).

$\Rightarrow$ (∗ Premise: σ is incompletely refined and Definition 5.3.4 (incomplete trace refinement) ∗)
$\exists \sigma_1, \ldots, \sigma_n \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge\, \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge\, (\sigma_n \cdot x \in \mathit{TXStraces}(r(\lambda_n)) \vee \sigma_n \cdot x \in \mathit{XStraces}(r(\lambda_n)))$
$\wedge\, \sigma_n \in \mathit{XStraces}(r(\lambda_n)) \wedge \sigma_n \neq \epsilon$

$\Rightarrow$ (∗ Definition 5.3.2 (*XStraces*) ϵ is explicitly excluded. ∗)
$\exists \sigma_1, \ldots, \sigma_n \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge\, \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge\, (\sigma_n \cdot x \in \mathit{TXStraces}(r(\lambda_n)) \vee \sigma_n \cdot x \in \mathit{XStraces}(r(\lambda_n)))$
$\wedge\, \sigma_n \in \mathit{XStraces}(r(\lambda_n)) \wedge \sigma_n \neq \epsilon$

$\Rightarrow$ (∗ Definition 5.3.1 (*TXStraces*), Definition 5.3.2 (*XStraces*) ∗)
$\exists \sigma_1, \ldots, \sigma_n \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge\, \sigma_n \in \mathit{XStraces}(r(\lambda_n)) \wedge \sigma \neq \epsilon \wedge r(\lambda_n) \xRightarrow{\sigma_n \cdot x}$
$\Rightarrow$ (∗ Definition *out* and **after**, note that $x \in U_{r\delta}$ ∗)
$\exists \sigma_1, \ldots, \sigma_n \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i))$
$\wedge\, \sigma_n \in \mathit{XStraces}(r(\lambda_n)) \wedge \sigma \neq \epsilon \wedge x \in \mathit{out}(r(\lambda_n)\ \textbf{after}\ \sigma_n)$
$\Rightarrow$ (∗ Premise: $\lambda_1 \cdots \lambda_n \in \mathit{Straces}(s)$ ∗)
$\exists \sigma_1, \ldots, \sigma_n \in L_{r\delta} : \sigma = \sigma_1 \cdots \sigma_n$
$\wedge \forall 1 \leq i < n : \sigma_i \in \mathit{TXStraces}(r(\lambda_i)) \wedge \sigma_n \in \mathit{XStraces}(r(\lambda_n))$
$\wedge\, \sigma \neq \epsilon \wedge x \in \mathit{out}(r(\lambda_n)\ \textbf{after}\ \sigma_n) \wedge \lambda_1 \cdots \lambda_n \in \mathit{Straces}(s)$
$\Rightarrow$ (∗ Definition 5.4.3 ∗)
$\sigma \in \mathit{out}_{inc}(s, \sigma, r)$

□

Theorem 5.4.7 Let $s \in \mathbf{LTS}(I, U), i \in \mathbf{IOTS}(I', U')$ with $(I' \cup U') \in L_r$.

$$i\ \mathbf{ioco}_r\ s \Leftrightarrow i\ \mathbf{ioco}\ s[r]$$

□

Proof We immediately expand the definitions of **ioco** and $\mathbf{ioco}_r$:
$\forall \sigma \in \mathit{Straces}(s)[r] : \mathit{out}(i\ \textbf{after}\ \sigma) \subseteq \mathit{out}_{rc}(s, \sigma, r) \cup \mathit{out}_{inc}(s, \sigma, r) \Leftrightarrow \forall \sigma' \in \mathit{Straces}(s[r]) : \mathit{out}(i\ \textbf{after}\ \sigma') \subseteq \mathit{out}(s[r]\ \textbf{after}\ \sigma')$

This follows from Theorem 5.3.12 and Proposition 5.4.6.

□

B.4 Proofs Section 5.5: Test-case refinement

The refinement function has the signature: $r : L_\tau \to \mathbf{FLTS}$. Because the refinement function is used almost everywhere in this document, we have chosen to omit it in the proofs of this section.

In order to proof Theorem 5.5.14 and Theorem 5.5.18 we have made several lemmas to help us. We start with several that deal with the relation between mini-tests and refinement transition systems. Next we treat several lemmas that proof relations between abstract and refined test cases.

The following lemma shows that for a trace of a refinement transition system, we can generate a mini-test that can perform the same trace.

Lemma B.4.1 Let $\lambda \in L_\delta, \sigma \in L_{r\delta}^*$

$$r(\lambda) \xRightarrow{\sigma} \text{implies } \exists m \in MT(r(\lambda)) : m \xrightarrow{\sigma}$$

□

Proof We strengthen the proof obligation in order to make the proof possible. We denote the set of mini-tests generated from Definition 5.5.2 with state set S as $\mathsf{TestGen}_{\mathsf{mt}}(S)$. S refers to the set of states in the mini-test generation algorithm (steps 3, 4 and 5 in Definition 5.5.2). Let $q \in Q_{r(\lambda)}$

$$r(\lambda) \overset{\sigma}{\Longrightarrow} q \Rightarrow \exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma) : m \xrightarrow{\sigma} t \qquad \text{(B.2)}$$
$$\wedge\, q \in (S \textbf{ after } \sigma)$$

Proof by induction on the length of σ.

Basic step: $\sigma = \epsilon$. Mini-tests do not have τ steps, therefore an ϵ transition for the mini-test is always τ^0.

$$\begin{array}{ll}
 & r(\lambda) \overset{\epsilon}{\Longrightarrow} q \\
\Rightarrow & (* \text{ Definition } \textbf{after}, \mathsf{start}_{r(\lambda)} \in S \ *) \\
 & r(\lambda) \overset{\epsilon}{\Longrightarrow} q \wedge q \in (S \textbf{ after } \epsilon) \\
\Rightarrow & (* \text{ Definition 5.5.2, we start with } S = \{\mathsf{start}_{r(\lambda)}, *\} \ *) \\
 & \exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \epsilon) : m \xrightarrow{\tau^0} t \\
 & \wedge\, q \in (S \textbf{ after } \epsilon)
\end{array}$$

Induction step: Let $\sigma = \sigma_1{\cdot}\lambda$ with $\sigma_1 \in L^*_{r\delta}$ and $\lambda \in L_{r\delta}$. Assume that the lemma holds for σ_1.

We identify the following cases:

- $\lambda \in I$

$$\begin{array}{ll}
 & r(\lambda) \overset{\sigma_1\cdot\lambda}{\Longrightarrow} q \\
\Rightarrow & (* \text{ Definition } \Longrightarrow \ *) \\
 & \exists q_1 \in Q_{r(\lambda)} : r(\lambda) \overset{\sigma_1}{\Longrightarrow} q_1 \overset{\lambda}{\Longrightarrow} q \\
\Rightarrow & (* \text{ Induction } *) \\
 & \exists q_1 \in Q_{r(\lambda)} : r(\lambda) \overset{\sigma_1}{\Longrightarrow} q_1 \overset{\lambda}{\Longrightarrow} q \wedge \exists m \in MT(r(\lambda)), \\
 & t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1) : m \xrightarrow{\sigma_1} t \wedge q_1 \in (S \textbf{ after } \sigma_1) \\
\Rightarrow & (* \text{ Definition 5.5.2 (mini-test generation rule 3:} \\
 & \quad q \in (q_1 \textbf{ after } \lambda)) \ *) \\
 & \exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1), \\
 & t' \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1{\cdot}\lambda) : m \xrightarrow{\sigma_1} t \xrightarrow{\lambda} t' \\
 & \wedge\, q \in (S \textbf{ after } \sigma_1{\cdot}\lambda) \\
\Rightarrow & (* \text{ Definition } \longrightarrow, \text{ logical reasoning } *) \\
 & \exists m \in MT(r(\lambda)), t' \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1{\cdot}\lambda) : m \xrightarrow{\sigma_1\cdot\lambda} t' \\
 & \wedge\, q \in (S \textbf{ after } \sigma_1{\cdot}\lambda) \\
\Rightarrow & (* \ \sigma = \sigma_1{\cdot}\lambda, \text{ rename } t' \text{ to } t \ *) \\
 & \exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma) : m \xrightarrow{\sigma} t \\
 & \wedge\, q \in (S \textbf{ after } \sigma)
\end{array}$$

- $\lambda \in U$

$r(\lambda) \xRightarrow{\sigma_1 \cdot \lambda} q$

$\Rightarrow$ (* Definition $\Longrightarrow$ *)

$\exists q_1 \in Q_{r(\lambda)} : r(\lambda) \xRightarrow{\sigma_1} q_1 \xRightarrow{\lambda} q$

$\Rightarrow$ (* Induction *)

$\exists q_1 \in Q_{r(\lambda)} : r(\lambda) \xRightarrow{\sigma_1} q_1 \xRightarrow{\lambda} q \wedge \exists m \in MT(r(\lambda)),$
$t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1) : m \xrightarrow{\sigma_1} t \wedge q_1 \in (S \textbf{ after } \sigma_1)$

$\Rightarrow$ (* Definition 5.5.2 (mini-test generation rule 4 and 5: $\lambda \in out(q_1)$, $q \in (q_1 \textbf{ after } \lambda)$) *)

$\exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1),$
$t' \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1 \cdot \lambda) : m \xrightarrow{\sigma_1} t \xrightarrow{\lambda} t'$
$\wedge\, q \in (S \textbf{ after } \sigma_1 \cdot \lambda)$

$\Rightarrow$ (* Definition $\longrightarrow$, logical reasoning *)

$\exists m \in MT(r(\lambda)), t' \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma_1 \cdot \lambda) : m \xrightarrow{\sigma_1 \cdot \lambda} t'$
$\wedge\, q \in (S \textbf{ after } \sigma_1 \cdot \lambda)$

$\Rightarrow$ (* $\sigma = \sigma_1 \cdot \lambda$, rename t' to t *)

$\exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma) : m \xrightarrow{\sigma} t$
$\wedge\, q \in (S \textbf{ after } \sigma)$

- $\lambda = \delta$. This proof is analogous to the proof of $\lambda \in U$.

□

Lemma B.4.2 Let $\lambda \in L_\delta$

$$\forall \sigma \in \lambda[r]_{rc}, \exists m \in MT(r(\lambda)) : m \xrightarrow{\sigma} \checkmark$$

□

Proof We use the stronger proof obligation B.2 of Lemma B.4.1

$\sigma \in \lambda[r]_{rc}$

$\Rightarrow$ (* Definition 5.3.3 (complete trace refinement) *)

$\sigma \in TXStraces(r(\lambda))$

$\Rightarrow$ (* Definition 5.3.1 ($TXStraces$) *)

$r(\lambda) \xRightarrow{\sigma} \mathsf{final}$

$\Rightarrow$ (* Result B.2 of the proof of Lemma B.4.1 *)

$\exists m \in MT(r(\lambda)), t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma) : m \xrightarrow{\sigma} t$
$\wedge\, \mathsf{final} \in (S \textbf{ after } \sigma)$

$\Rightarrow$ (* Definition 5.5.2 rule 2 *)

$\exists m \in MT(r(\lambda)) : m \xrightarrow{\sigma} \checkmark$

□

Lemma B.4.3 Let $\lambda \in L_\delta$

$$\forall \sigma \in \lambda[r]_{inc}, \exists m \in MT(r(\lambda)) : m \xrightarrow{\sigma}$$

□

Proof This proof follows directly from Lemma B.4.1 $\square$

Lemma B.4.4 Let $t \in \mathbf{TEST}(I,U), q_1, q_2 \in Q_t \backslash \mathsf{Fail}_t, \lambda \in L_\delta, \sigma \in L^*_{r\delta} \backslash \{\epsilon\}$

$$q_1 \xrightarrow{\lambda} q_2 \wedge \exists m \in MT(r(\lambda)), q_2' \in Q_m : m \xrightarrow{\sigma} q_2' \wedge \exists f : f_{q_2}(\lambda) = m$$
$$\text{implies } (q_1, \checkmark) \xrightarrow{\sigma}_{t[f]} (q_2, q_2')$$

$\square$

Proof The case $\sigma = \epsilon$ is excluded because the lemma does not hold for this case. The reason is that test cases do not have τ steps, which implies that in the lemma σ consists of zero τ steps. As a result q_2' is the start state of the mini-test, but $q_2' \neq \checkmark$.

For the proof we split σ into n actions: $\sigma = \lambda_1 \cdots \lambda_n$ for some $n \geq 1$ and $\forall 1 \leq i \leq n : \lambda_i \in L_{r\delta}$. To keep the notation concise, we write $q_2' = t_n$.

$$\begin{array}{ll}
 & q_1 \xrightarrow{\lambda} q_2 \wedge \exists n \geq 1, m \in MT(r(\lambda)), t_1 \cdots t_n \in Q_m : \\
 & m \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \\
\Rightarrow & (*\text{ Logical reasoning } *) \\
 & q_1 \xrightarrow{\lambda} q_2 \wedge \exists n \geq 1, m \in MT(r(\lambda)), t_1 \cdots t_n \in Q_m : \\
 & m \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \wedge \exists (f : Q_t \to L_\delta \to MT) : f_{q_2}(\lambda) = m \\
\Rightarrow & (*\text{ Definition 5.5.5 (test case refinement: } T_1) \quad *) \\
 & q_1 \xrightarrow{\lambda} q_2 \wedge \exists n \geq 1, m \in MT(r(\lambda)), t_1 \cdots t_n \in Q_m : \\
 & t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \\
 & \wedge \exists (f : Q_t \to L_\delta \to MT) : f_{q_2}(\lambda) = m \wedge (q_1, \checkmark) \xrightarrow{\lambda_1}_{t[f]} (q_2, t_1) \\
\Rightarrow & (*\text{ Definition 5.5.5 (test case refinement: } T_2) \quad *) \\
 & \exists (f : Q_t \to L_\delta \to MT) : (q_1, \checkmark) \xrightarrow{\lambda_1}_{t[f]} (q_2, t_1) \\
 & \wedge \forall 1 \leq i < n : (q_2, t_i) \xrightarrow{\lambda_i}_{t[f]} (q_2, t_{i+1})
\end{array}$$

This step makes use of the property of mini-tests that $\checkmark$ is a final state in a mini-test. This means that none of the t_i states equals $\checkmark$.

$$\begin{array}{ll}
\Rightarrow & (*\text{ Definition 5.5.11 (test case refinement) } *) \\
 & \exists t_r \in t[r] : (q_1, \checkmark) \xrightarrow{\lambda_1}_r (q_2, t_1) \cdots (q_2, t_{n-1}) \xrightarrow{\lambda_n}_r (q_2, q_n) \\
\Rightarrow & (*\text{ Definition } \longrightarrow\text{, logical reasoning, premise: } q_n = q_2' \quad *) \\
 & \exists t_r \in t[r] : (q_1, \checkmark) \xrightarrow{\sigma}_r (q_2, q_2')
\end{array}$$

$\square$

Lemma B.4.5 Let $t \in \mathbf{TEST}(I,U), q \notin \mathsf{Fail}_t, \sigma \in L^*_{r\delta}$

$$t \xrightarrow{\sigma} q \Rightarrow \forall \sigma' \in \sigma[r]_{rc}, \exists t_r \in t[r] : t_r \xrightarrow{\sigma'}_r (q, \checkmark)$$

$\square$

Proof We treat the case $\sigma = \epsilon$ separate, because it is slightly different. Test-cases do not have τ-steps, therefore ϵ means zero τ-steps. This means that $q = \mathsf{start}$. $(\mathsf{start}, \checkmark)$ is the start state of t_r.

$t \xrightarrow{\tau^0} q \wedge q = \mathsf{start}$

$\Rightarrow$ (∗ Logical reasoning ∗)

$(\mathsf{start}, \checkmark) \xrightarrow{\tau^0} (\mathsf{start}, \checkmark)$

$\Rightarrow$ (∗ Definition 5.5.5 $(\mathsf{start}, \checkmark)$ is the start state of t_r ∗)

$t_r \xrightarrow{\tau^0} (\mathsf{start}, \checkmark)$

For $\sigma \neq \epsilon$, we split σ into $\lambda_1 \cdots \lambda_n$ for some $n \geq 1$ such that $\sigma = \lambda_1 \cdots \lambda_n$ and $\forall 1 \leq i \leq n : \lambda_i \in L_{r\delta}$. To keep the notation concise, we write $t = t_0$ and $q = t_n$.

$t_0 \xrightarrow{\lambda_1 \cdots \lambda_n} t_n$

$\Rightarrow$ (∗ Definition of $\longrightarrow$ ∗)

$\exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n$

$\Rightarrow$ (∗ Lemma B.4.2 ∗)

$\exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n$

$\wedge \forall 1 \leq i \leq n, \sigma_i \in \lambda_i[r]_{rc}, \exists m \in MT(r(\lambda_i)) : m \xrightarrow{\sigma_i} \checkmark$

$\Rightarrow$ (∗ Logical reasoning: a priori we do not exclude any function ∗)

$\exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n$

$\wedge \forall 1 \leq i \leq n, \sigma_i \in \lambda_i[r]_{rc}, \exists m_i \in MT(r(\lambda_i)) : m_i \xrightarrow{\sigma_i} \checkmark$

$\wedge \exists (f : Q_t \rightarrow L_\delta \rightarrow MT) : f_{t_i}(\lambda_i) = m_i$

$\Rightarrow$ (∗ Lemma B.4.4 ∗)

$\exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n$

$\wedge \forall 1 \leq i \leq n, \sigma_i \in \lambda_i[r]_{rc}, \exists m_i \in MT(r(\lambda_i)) : m_i \xrightarrow{\sigma_i} \checkmark$

$\wedge \exists (f : Q_t \rightarrow L_\delta \rightarrow MT) : f_{t_i}(\lambda_i) = m_i \wedge (t_{i-1}, \checkmark) \xrightarrow{\sigma_i}_{t[f]} (t_i, \checkmark)$

Note that Lemma B.4.4 is not defined for subtraces σ_i equal to ϵ. This is no problem for this proof, because $\sigma_i \in \lambda_i[r]_{rc}$ with $\lambda_i \in L_{r\delta}$. ϵ is only a completely refined trace of $\tau[r]_{rc}$.

$\Rightarrow$ (∗ Logical reasoning ∗)

$\exists t_0, \cdots, t_n \in Q_t, \forall 1 \leq i \leq n, \sigma_i \in \lambda_i[r]_{rc}, \exists (f : Q_t \rightarrow L_\delta \rightarrow MT) :$
$(t_0, \checkmark) \xrightarrow{\sigma_1}_{t[f]} (t_1, \checkmark) \cdots (t_{n-1}, \checkmark) \xrightarrow{\sigma_n}_{t[f]} (t_n, \checkmark)$

$\Rightarrow$ (∗ Definition $\longrightarrow$ ∗)

$\forall 1 \leq i \leq n, \sigma_i \in \lambda_i[r]_{rc}, \exists (f : Q_t \rightarrow L_\delta \rightarrow MT) :$
$(t_0, \checkmark) \xrightarrow{\sigma_1 \cdots \sigma_n}_{t[f]} (t_n, \checkmark)$

$\Rightarrow$ (∗ Lemma B.1.3 (trace refinement is compositional) ∗)

$\forall \sigma' \in \sigma[r]_{rc}, \exists (f : Q_t \rightarrow L_\delta \rightarrow MT) : (t_0, \checkmark) \xrightarrow{\sigma}_{t[f]} (t_n, \checkmark)$

$\Rightarrow$ (∗ Definition 5.5.11 (test case refinement) ∗)

$\forall \sigma' \in \sigma[r]_{rc}, \exists t_r \in t[r] : (t_0, \checkmark) \xrightarrow{\sigma}_r (t_n, \checkmark)$

$\Rightarrow$ (∗ Logical reasoning: $t = t_0$, $q = t_n$ ∗)

$\forall \sigma' \in \sigma[r]_{rc}, \exists t_r \in t[r] : t_r \xrightarrow{\sigma}_r (q, \checkmark)$

□

Lemma B.4.6 Let $t \in \mathbf{TEST}(I, U), q \notin \mathsf{Fail}_t$

$$t \xrightarrow{\sigma} q \Rightarrow \forall \sigma' \in \sigma[r]_{inc}, \exists t_r \in t[r], q' \notin L_{r\delta} : t_r \xrightarrow{\sigma'}_r (q, q')$$

□

$q' \notin L_{r\delta}$ expresses that the state (q, q') is not a state that is used in the T_3 transitions of the definition of test case refinement.

Proof We treat the case $\sigma = \epsilon$ separately. Incomplete trace refinement of ϵ yields the empty set, rendering the lemma vacuously true.

For $\sigma \neq \epsilon$ we split σ into $\lambda_1 \cdots \lambda_n$ for some $n \geq 1$ such that $\sigma = \lambda_1 \cdots \lambda_n$ and $\forall 1 \leq i \leq n : \lambda_i \in L_{r\delta}$. To keep the notation concise we write $t = q_0$ and $q = t_n$.

$$\begin{array}{ll}
 & t_0 \xrightarrow{\lambda_1 \cdots \lambda_n} t_n \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists t_0, \cdots, t_n : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \\
\Rightarrow & (*\ \text{Lemma B.4.2}\ *) \\
 & \exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \\
 & \wedge \forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \exists m_i \in MT(r(\lambda_i)) : m_i \xrightarrow{\sigma_i} \checkmark \\
\Rightarrow & (*\ \text{Lemma B.4.3}\ *) \\
 & \exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \\
 & \wedge \forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \exists m_i \in MT(r(\lambda_i)) : m_i \xrightarrow{\sigma_i} \checkmark \\
 & \wedge \forall \sigma_n \in \lambda_n[r]_{inc}, \exists m_n \in MT(r(\lambda_n)), q' \in Q_{m_n} : m_n \xrightarrow{\sigma_n} q' \\
\Rightarrow & (*\ \text{Logical reasoning: a priori we do not exclude any function}\ *) \\
 & \exists t_0, \cdots, t_n \in Q_t : t_0 \xrightarrow{\lambda_1} t_1 \cdots t_{n-1} \xrightarrow{\lambda_n} t_n \\
 & \wedge \forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \exists m \in MT(r(\lambda_i)) : m_i \xrightarrow{\sigma_i} \checkmark \\
 & \wedge \forall \sigma_n \in \lambda_n[r]_{inc}, \exists m_n \in MT(r(\lambda_n)), \exists q' \in Q_{m_n} : m_n \xrightarrow{\sigma_n} q' \\
 & \wedge \forall 1 \leq i \leq n, \exists (f : Q_t \to L_\delta \to MT) : f_{t_i}(\lambda_i) = m_i
\end{array}$$

$$\begin{array}{ll}
\Rightarrow & (*\ \text{Lemma B.4.4}, \forall 1 \leq i \leq n : t_i \notin \mathsf{Fail}_t\ *) \\
 & \forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \exists m_i \in MT(r(\lambda_i)) : m_i \xrightarrow{\sigma_i} \checkmark \\
 & \forall \sigma_n \in \lambda_n[r]_{inc}, \exists m_n \in MT(r(\lambda_n)), q' \in Q_{m_n} : m_n \xrightarrow{\sigma_n} q' \\
 & \wedge \exists (f : Q_t \to L_\delta \to MT) : (t_{i-1}, \checkmark) \xrightarrow{\sigma_i}_{t[f]} (t_i, \checkmark) \\
 & \wedge (t_{n-1}, \checkmark) \xrightarrow{\sigma_n}_{t[f]} (t_n, q')
\end{array}$$

Note that Lemma B.4.4 is not defined for subtraces σ_i equal to ϵ. This is no problem for this proof, because $\sigma_i \in \lambda_i[r]_{rc}$ with $\lambda_i \in L_{r\delta}$. ϵ is only a completely refined trace of $\tau[r]_{rc}$. ϵ is never the result of an incompletely refined trace.

We abbreviate $\exists (f : Q_t \to L_\delta \to MT)$ to $\exists f$.

$\Rightarrow$ (* Definition 5.5.5, states mini-tests not in $L_{r\delta}$ *)
$\forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \exists m \in MT(r(\lambda_i)) : m \xrightarrow{\sigma_i} \checkmark$
$\wedge \forall \sigma_n \in \lambda_n[r]_{inc}, \exists m_1 \in MT(r(\lambda_n)), q' \notin L_{r\delta} : m_1 \xrightarrow{\sigma_n} q'$
$\wedge \exists f : (t_{i-1}, \checkmark) \xrightarrow{\sigma_i}_{t[f]} (t_i, \checkmark) \wedge (t_{n-1}, \checkmark) \xrightarrow{\sigma_n}_{t[f]} (t_n, q')$
$\Rightarrow$ (* Logical reasoning *)
$\forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \sigma_n \in \lambda_n[r]_{inc}, \exists f, q' \notin L_{r\delta} :$
$(t_0, \checkmark) \xrightarrow{\sigma_1}_{t[f]} (t_1, \checkmark) \cdots (t_{n-1}, \checkmark) \xrightarrow{\sigma_n}_{t[f]} (t_n, q')$
$\Rightarrow$ (* Definition $\longrightarrow$ *)
$\forall 1 \leq i < n, \sigma_i \in \lambda_i[r]_{rc}, \sigma_n \in \lambda_n[r]_{inc}, \exists f, q' \notin L_{r\delta} :$
$(t_0, \checkmark) \xrightarrow{\sigma_1 \cdots \sigma_n}_{t[f]} (t_n, q')$
$\Rightarrow$ (* Lemma B.1.4 *)
$\forall \sigma' \in \sigma[r]_{inc}, \exists f, q' \notin L_{r\delta} : (t_0, \checkmark) \xrightarrow{\sigma}_{t[f]} (t_n, q')$
$\Rightarrow$ (* Definition 5.5.11 *)
$\forall \sigma' \in \sigma[r]_{inc}, \exists t_r \in t[r], q' \notin L_{r\delta} : (t_0, \checkmark) \xrightarrow{\sigma}_{r} (t_n, q')$
$\Rightarrow$ (* Logical reasoning, $t = t_0$ and $q = t_n$ *)
$\forall \sigma' \in \sigma[r]_{inc}, \exists t_r \in t[r], q' \notin L_{r\delta} : t_r \xrightarrow{\sigma}_{r} (q, q')$

□

Lemma B.4.7 Let $t \in \mathbf{TEST}(I, U), q \notin \mathsf{Fail}$

$$t \xrightarrow{\sigma} q \Rightarrow \forall \sigma' \in \sigma[r], \exists t_r \in t[r], q' \in Q_{t_r} : t_r \xrightarrow{\sigma'}_{r} q'$$

□

Proof This lemma follows directly from Lemma B.4.5 and Lemma B.4.6.

□

Lemma B.4.8 Let $\sigma \in L_{r\delta}^*, m \in MT(r(\lambda)), q \in Q_m \backslash \{\checkmark\}$

$$m \xrightarrow{\sigma} q \Rightarrow \sigma \in \lambda[r]_{inc}$$

□

Proof Analogous to the proof of Lemma B.4.1 we strengthen our proof obligation to make the proof possible. We introduce the set of states $S = \{*, \mathsf{start}\}$ from the mini-test generation algorithm (Definition 5.5.2). With $\mathsf{TestGen}_{\mathsf{mt}}(S)$, we denote a mini-test generated from Definition 5.5.2 with state set S. Let $t \neq \checkmark$.

$$m \xrightarrow{\sigma} t \Rightarrow \exists q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma}{\Longrightarrow} q \wedge q \in (S \textbf{ after } \sigma) \wedge t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma) \quad \text{(B.3)}$$

Proof by induction on the length of σ

Basic step: $\sigma = \epsilon$. This is a special case, as mini-tests do not have τ steps. We start the algorithm with $S = \{\mathsf{start}_{r(\lambda)}, *\}$. $*$ is a pseudo state to prevent δ observations in the start state. For τ^0 the set of states S does not change and $\mathsf{start}_{r(\lambda)} \in S$ **after** ϵ.

$$\begin{array}{ll}
 & m \xrightarrow{\epsilon} t \\
\Rightarrow & (*\ \text{Definition 5.5.2}\ *) \\
 & r(\lambda) \overset{\epsilon}{\Longrightarrow} \mathsf{start}_{r(\lambda)} \wedge \mathsf{start}_{r(\lambda)} \in (S\ \mathbf{after}\ \epsilon) \\
 & \wedge\, t \in \mathsf{TestGen}_{\mathsf{mt}}(S\ \mathbf{after}\ \epsilon)
\end{array}$$

Induction step: Let $\sigma = \sigma' \cdot \lambda'$ with $\sigma' \in L^*_{r\delta}$ and $\lambda' \in L_{r\delta}$ and assume that the lemma holds for σ'. We identify the following cases:

- $\lambda' \in I$

$$\begin{array}{ll}
 & m \xrightarrow{\sigma'\cdot\lambda'} t \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists t' : m \xrightarrow{\sigma'} t' \xrightarrow{\lambda'} t \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists t' \in \mathsf{TestGen}_{\mathsf{mt}}(S\ \mathbf{after}\ \sigma'), q_1 \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : \\
 & m \xrightarrow{\sigma'} t' \xrightarrow{\lambda'} t \wedge r(\lambda) \overset{\sigma'}{\Longrightarrow} q_1 \wedge q_1 \in (S\ \mathbf{after}\ \sigma') \\
\Rightarrow & (*\ \text{Definition 5.5.2 (mini-test generation) rule 3}\ *) \\
 & \exists q_1, q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma'}{\Longrightarrow} q_1 \overset{\lambda'}{\Longrightarrow} q \\
 & \wedge\, q \in (S\ \mathbf{after}\ \sigma'\cdot\lambda') \wedge t \in \mathsf{TestGen}_{\mathsf{mt}}(S\ \mathbf{after}\ \sigma'\cdot\lambda') \\
\Rightarrow & (*\ \text{Definition} \Longrightarrow\ *) \\
 & \exists q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma'\cdot\lambda'}{\Longrightarrow} q \wedge q \in (S\ \mathbf{after}\ \sigma'\cdot\lambda') \\
 & \wedge\, t \in \mathsf{TestGen}_{\mathsf{mt}}(S\ \mathbf{after}\ \sigma'\cdot\lambda') \\
\Rightarrow & (*\ \sigma = \sigma'\cdot\lambda'\ *) \\
 & \exists q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma}{\Longrightarrow} q \wedge q \in (S\ \mathbf{after}\ \sigma) \\
 & \wedge\, t \in \mathsf{TestGen}_{\mathsf{mt}}(S\ \mathbf{after}\ \sigma)
\end{array}$$

- $\lambda' \in U$

$$\begin{array}{ll}
 & m \xrightarrow{\sigma'\cdot\lambda'} t \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists t' : m \xrightarrow{\sigma'} t' \xrightarrow{\lambda'} t \\
\Rightarrow & (*\ \text{Induction}\ *) \\
 & \exists t' \in \mathsf{TestGen}_{\mathsf{mt}}(S\ \mathbf{after}\ \sigma'), q_1 \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : \\
 & m \xrightarrow{\sigma'} t' \xrightarrow{\lambda'} t \wedge r(\lambda) \overset{\sigma'}{\Longrightarrow} q_1 \wedge q_1 \in (S\ \mathbf{after}\ \sigma')
\end{array}$$

$\Rightarrow$ (∗ Definition 5.5.2 (mini-test generation) rule 4 and 5 ∗)
$\exists q_1, q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma'}{\Longrightarrow} q_1 \overset{\lambda'}{\Longrightarrow} q$
$\wedge\, q \in (S \textbf{ after } \sigma' \cdot \lambda') \wedge t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma' \cdot \lambda')$
$\Rightarrow$ (∗ Definition $\Longrightarrow$ ∗)
$\exists q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma' \cdot \lambda'}{\Longrightarrow} q \wedge q \in (S \textbf{ after } \sigma' \cdot \lambda')$
$\wedge\, t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma' \cdot \lambda')$
$\Rightarrow$ (∗ $\sigma = \sigma' \cdot \lambda'$ ∗)
$\exists q \in Q_{r(\lambda)} \backslash \{\mathsf{final}_{r(\lambda)}\} : r(\lambda) \overset{\sigma}{\Longrightarrow} q \wedge q \in (S \textbf{ after } \sigma)$
$\wedge\, t \in \mathsf{TestGen}_{\mathsf{mt}}(S \textbf{ after } \sigma)$

- $\lambda = \delta$, this case is analogous to the previous step. Note that δ is by definition not allowed if $* \in S$.

From this intermediate result we can conclude, using Definition 5.3.4 (incomplete trace refinement), that $\sigma \in \lambda[r]_{inc}$. □

Lemma B.4.9 Let $m \in MT(r(\lambda))$

$$m \xrightarrow{\sigma} \checkmark \Rightarrow \sigma \in \lambda[r]_{rc}$$

□

Proof Analogous to the proof of Lemma B.4.8 we obtain for the case $q = \checkmark$ that $r(\lambda) \overset{\sigma}{\Longrightarrow} \mathsf{final}_{r(\lambda)}$. Using Definition 5.3.3 (complete trace refinement) we know that $\sigma \in \lambda[r]_{rc}$

□

Many of the previous lemmas showed properties of refined test-cases, based on properties of an abstract test-case. The following lemmas are the other way around. They show that properties of a refined test-case imply certain properties of the abstract test-case.

The following lemma depends on a property that is very similar to the final state clean property for transition systems. The lemma expresses that for traces that do not encounter states with ✓ as the second state element, originate from one mini-test.

Lemma B.4.10 Let $(q_1, \checkmark), (q, q') \in Q_r, q_1, q \notin \mathsf{Fail}, q' \notin L_{r\delta}, \sigma \in L^*_{r\delta}$ and the trace σ does not encounter any intermediate states with ✓ as the second state element.

$$(q_1, \checkmark) \xrightarrow{\sigma}_r (q, q') \wedge (q_1, \checkmark) \neq (q, q')$$
$$\Rightarrow \exists \lambda \in L_\delta : q_1 \xrightarrow{\lambda} q \wedge \exists m \in MT(\lambda) : m \xrightarrow{\sigma} q'$$

□

Proof Test cases do not have τ steps, therefore the only possibility for ϵ is zero τ-steps. This violates the premise: $(q_1, \checkmark) \neq (q, q')$. This means that σ consists of one or more actions. We split σ into $\lambda_1 \cdots \lambda_n$ such that $\forall 1 \leq i \leq n : \lambda_i \in L_{r\delta}$ and $\sigma = \lambda_1 \cdots \lambda_n$. In the proof we use $(q, q') = (q_{n+1}, q'_{n+1})$.

$$
\begin{array}{ll}
& (q_1, \checkmark) \xrightarrow{\lambda_1 \cdots \lambda_n}_r (q, q') \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
& \forall 1 \leq i \leq n, \exists (q_i, q'_i) \in Q_r : \\
& (q_1, \checkmark) \xrightarrow{\lambda_1}_r (q_2, q'_2) \cdots (q_n, q'_n) \xrightarrow{\lambda_n}_r (q_{n+1}, q'_{n+1}) \\
\Rightarrow & (*\ \text{Definition 5.5.5 (test case refinement: } T_2\text{), premise } q \notin \mathsf{Fail} \\
& \quad \text{and no intermediate } \checkmark \text{ states } *) \\
& \forall 1 \leq i \leq n, \exists (q_i, q'_i) \in Q_r : \\
& (q_1, \checkmark) \xrightarrow{\lambda_1}_r (q_2, q'_2) \cdots (q_n, q'_n) \xrightarrow{\lambda_n}_r (q_{n+1}, q'_{n+1}) \\
& \wedge \forall 1 < i \leq n, \exists \lambda \in L_\delta, (f : Q \to L_\delta \to MT), p \in Q_t : \\
& (p, \lambda, q_i) \in T \wedge (q'_i, \lambda_i, q'_{i+1}) \in T_{f_{q_i}(\lambda)} \wedge q_i = q_{i+1} = q \\
\Rightarrow & (*\ \text{Logical reasoning: } q = q_i\ *) \\
& \forall 1 \leq i \leq n, \exists (q_i, q'_i) \in Q_r : \\
& (q_1, \checkmark) \xrightarrow{\lambda_1}_r (q, q'_2) \cdots (q, q'_n) \xrightarrow{\lambda_n}_r (q, q'_{n+1}) \\
& \wedge \forall 1 < i \leq n, \exists \lambda \in L_\delta, (f : Q \to L_\delta \to MT), p \in Q : \\
& (p, \lambda, q) \in T \wedge (q'_i, \lambda_i, q'_{i+1}) \in T_{f_q(\lambda)} \\
\Rightarrow & (*\ \text{Definition 5.5.5 (test case refinement: } T_1\text{), premise } q \notin \mathsf{Fail}\ *) \\
& \forall 1 \leq i \leq n, \exists (q_i, q'_i) \in Q_r : \\
& (q_1, \checkmark) \xrightarrow{\lambda_1}_r (q, q'_2) \cdots (q, q'_n) \xrightarrow{\lambda_n}_r (q, q'_{n+1}) \\
& \wedge \forall 1 < i \leq n, \exists \lambda \in L_\delta, (f : Q \to L_\delta \to MT), p \in Q : \\
& (p, \lambda, q) \in T \wedge (q'_i, \lambda_i, q'_{i+1}) \in T_{f_q(\lambda)} \wedge \exists \lambda' \in L_\delta : q_1 \xrightarrow{\lambda'} q \\
& \wedge \exists (f' : Q \to L_\delta \to MT) : (\mathsf{start}_{f'_q(\lambda)}, \lambda_1, q'_2) \in T_{f'_q(\lambda)} \\
\Rightarrow & (*\ \text{Logical reasoning: states of refinement transition systems} \\
& \quad \text{are unique. Therefore } \lambda' = \lambda \text{ and } f' = f\ *) \\
& \exists \lambda \in L_\delta : q_1 \xrightarrow{\lambda} q \\
& \wedge \exists (f : Q \to L_\delta \to MT) : (\mathsf{start}_{f_q(\lambda)}, \lambda_1, q'_2) \in T_{f_q(\lambda)} \\
& \wedge \forall 1 < i \leq n, \exists (q_i, q'_i) \in Q_r : (q'_i, \lambda_i, q'_{i+1}) \in T_{f_q(\lambda)} \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
& \exists \lambda \in L_\delta : q_1 \xrightarrow{\lambda} q \wedge \exists (f : Q \to L_\delta \to MT) : \mathsf{start}_{f_q(\lambda)} \xrightarrow{\lambda_1 \cdots \lambda_n}_{f_q(\lambda)} q \\
\Rightarrow & (*\ \text{Logical reasoning: } f \text{ has a mini-test as result, } q_{n+1} = q\ *) \\
& \exists \lambda \in L_\delta : q_1 \xrightarrow{\lambda} q \wedge \exists m \in MT(\lambda) : m \xrightarrow{\lambda_1 \cdots \lambda_n}_m q
\end{array}
$$

Although not explicitly stated, transitions from T_3 are not possible, because $q' \notin L_{r\delta}$.

□

Lemma B.4.11 Let $t \in \mathbf{TEST}(I, U), t_r \in t[r], q, q' \notin \mathsf{Fail}_t$

$$(q, \checkmark) \xrightarrow{\sigma}_r (q', \checkmark) \Rightarrow \exists \sigma' \in L_\delta^* : \sigma \in \sigma'[r]_{rc} \wedge q \xrightarrow{\sigma'} q'$$

□

Proof From the definition of test case refinement we see that $\exists (f : Q_t \to L_\delta \to MT) : t_r \in t[f]$. $f_q(\lambda)$ gives a mini-test for refinement transition system $r(\lambda)$.

Proof by induction on the number of intermediate state pairs that have $\checkmark$ as their second state. Let n denote the number of such state pairs.

Basic step: $n = 0$. This proof holds by construction. We distinguish two cases

- $\sigma = \epsilon$. This case holds straight forward as there are no τ steps in test cases.

$$\begin{array}{ll} & (q, \checkmark) \xrightarrow{\epsilon}_r (q', \checkmark) \\ \Rightarrow & (*\ \text{Definition } \epsilon\text{, in this case zero } \tau\text{-steps}\ *) \\ & q = q' \wedge q \xrightarrow{\epsilon} q' \\ \Rightarrow & (*\ \text{Definition 5.3.5}\ *) \\ & \epsilon \in \epsilon[r] \wedge q \xrightarrow{\epsilon} q' \end{array}$$

- $|\sigma| > 0$. Because $\sigma \neq \epsilon$ we know that $(q, \checkmark) \neq (q', \checkmark)$.

$$\begin{array}{ll} & (q, \checkmark) \xrightarrow{\sigma}_r (q', \checkmark) \\ \Rightarrow & (*\ \text{Lemma B.4.10}\ *) \\ & \exists \lambda \in L_\delta : q \xrightarrow{\lambda} q' \wedge \exists m \in MT(\lambda) : m \xrightarrow{\sigma} \checkmark \\ \Rightarrow & (*\ \text{Lemma B.4.9}\ *) \\ & \exists \lambda \in L_\delta : q \xrightarrow{\lambda} q' \wedge \sigma \in \lambda[r]_{rc} \end{array}$$

Induction step: Suppose that σ passes $n+1$ $\checkmark$-states and assume that the lemma holds for $n \geq 0$. For the proof we split σ into σ_1 and σ_2 in such a way that σ_1 passes n $\checkmark$ states. This means that σ_2 does not pass any $\checkmark$ states (together σ_1 and σ_2 pass $n+1$ $\checkmark$ states).

$$\begin{array}{ll} & (q, \checkmark) \xrightarrow{\sigma_1 \cdot \sigma_2}_r (q', \checkmark) \\ \Rightarrow & (*\ \text{Definition } \longrightarrow\ *) \\ & \exists (q_1, \checkmark) \in Q_{t_r} : (q, \checkmark) \xrightarrow{\sigma_1}_r (q_1, \checkmark) \xrightarrow{\sigma_2}_r (q', \checkmark) \\ \Rightarrow & (*\ \text{Induction}\ *) \\ & \exists (q_1, \checkmark) \in Q_{t_r} : (q_1, \checkmark) \xrightarrow{\sigma_2}_r (q', \checkmark) \\ & \wedge \exists \sigma_1' \in L_\delta^* : \sigma_1 \in \sigma_1'[r]_{rc} \wedge q \xrightarrow{\sigma_1'} q_1 \\ \Rightarrow & (*\ \text{Basic step (no intermediate } \checkmark \text{ states for } \sigma_2)\ *) \\ & \exists (q_1, \checkmark) \in Q_{t_r}, \sigma_1' \in L_\delta^* : \sigma_1 \in \sigma_1'[r]_{rc} \wedge q \xrightarrow{\sigma_1'} q_1 \\ & \wedge \exists \sigma_2' \in L_\delta^* : \sigma_2 \in \sigma_2'[r]_{rc} \wedge q_1 \xrightarrow{\sigma_2'} q' \\ \Rightarrow & (*\ \text{Definition } \longrightarrow\ *) \\ & \exists \sigma_1', \sigma_2' \in L_\delta^* : \sigma_1 \in \sigma_1'[r]_{rc} \wedge \sigma_2 \in \sigma_2'[r]_{rc} \wedge q \xrightarrow{\sigma_1' \cdot \sigma_2'} q' \\ \Rightarrow & (*\ \text{Lemma B.1.3}\ *) \\ & \exists \sigma_1', \sigma_2' \in L_\delta^* : (\sigma_1 \cdot \sigma_2) \in (\sigma_1' \cdot \sigma_2')[r]_{rc} \wedge q \xrightarrow{\sigma_1' \cdot \sigma_2'} q' \\ \Rightarrow & (*\ \text{Logical reasoning } (\sigma = \sigma_1 \cdot \sigma_2\text{, rewrite } \sigma_1' \cdot \sigma_2' \text{ to } \sigma')\ *) \\ & \exists \sigma' \in L_\delta^* : \sigma \in \sigma'[r]_{rc} \wedge q \xrightarrow{\sigma'} q' \end{array}$$

□

Lemma B.4.12 Let $t \in \mathbf{TEST}(I,U), t_r \in t[r], (q_1, \checkmark), (q, q') \in Q_r, q, q_1 \notin \mathsf{Fail}_t, q' \neq \checkmark, q' \notin L_{r\delta}$

$$(q_1, \checkmark) \xrightarrow{\sigma}_r (q, q') \Rightarrow \exists \sigma' \in L_\delta^* : \sigma \in \sigma'[r]_{inc} \wedge q_1 \xrightarrow{\sigma'} q$$

□

Proof For this proof we want to split σ into σ_1 and σ_2 such that σ_2 does not encounter any $\checkmark$ states for the second state of the state pair. We identify the case where σ encounters zero $\checkmark$ states and the case where σ encounters one or more $\checkmark$ states. We start with the case where the number of intermediate $\checkmark$ states is zero.

$(q_1, \checkmark) \xrightarrow{\sigma} (q, q')$
$\Rightarrow$ $(*$ Premise: $q' \neq \checkmark$ $*)$
$(q_1, \checkmark) \xrightarrow{\sigma} (q, q') \wedge (q_1, \checkmark) \neq (q, q')$
$\Rightarrow$ $(*$ Lemma B.4.10, $q_1, q \notin \mathsf{Fail}_t$, $q' \notin L_{r\delta}$ $*)$
$\exists \lambda \in L_\delta : q_1 \xrightarrow{\lambda} q \wedge \exists m \in MT(r(\lambda)) : m \xrightarrow{\sigma} q'$
$\Rightarrow$ $(*$ Lemma B.4.8, note that $q' \neq \checkmark$ $*)$
$\exists \lambda \in L_\delta : q_1 \xrightarrow{\lambda} q \wedge \sigma \in \lambda[r]_{inc}$

The number of intermediate states (q_2, q_2') that σ passes with $q_2' = \checkmark$ is greater than zero. This means that we can split σ into two parts σ_1 and σ_2 where σ_2 does not encounter any intermediate $\checkmark$ states (as a result σ_1 encounters fewer intermediate states than σ).

$\exists (q_2, \checkmark) \in Q_r : (q_1, \checkmark) \xrightarrow{\sigma_1} (q_2, \checkmark) \xrightarrow{\sigma_2} (q, q')$
$\Rightarrow$ $(*$ First case in this proof $*)$
$\exists (q_2, \checkmark) \in Q_r : (q_1, \checkmark) \xrightarrow{\sigma_1} (q_2, \checkmark) \xrightarrow{\sigma_2} (q, q')$
$\wedge \exists \lambda \in L_\delta : q_2 \xrightarrow{\lambda} q \wedge \sigma_2 \in \lambda[r]_{inc}$

Because $q' \neq \checkmark$ we know that $\sigma_2 \neq \epsilon$. By Definition 5.5.5 we know that $q_2 \notin \mathsf{Fail}_t$, because it has an outgoing transition.

$\Rightarrow$ $(*$ Lemma B.4.11 $*)$
$\exists (q_2, \checkmark) \in Q_r : (q_1, \checkmark) \xrightarrow{\sigma_1} (q_2, \checkmark) \xrightarrow{\sigma_2} (q, q')$
$\wedge \exists \lambda \in L_\delta : q_2 \xrightarrow{\lambda} q \wedge \sigma_2 \in \lambda[r]_{inc}$
$\wedge \exists \sigma_1' \in L_\delta^* : \sigma_1 \in \sigma_1'[r]_{rc} \wedge q_1 \xrightarrow{\sigma_1'} q_2$
$\Rightarrow$ $(*$ Definition $\longrightarrow$ $*)$
$\exists \sigma_1' \in L_\delta^*, \lambda \in L_\delta : q_1 \xrightarrow{\sigma_1' \cdot \lambda} q \wedge \sigma_1 \in \sigma_1'[r]_{rc} \wedge \sigma_2 \in \lambda[r]_{inc}$
$\Rightarrow$ $(*$ Lemma B.1.4 $*)$
$\exists \sigma_1' \in L_\delta^*, \lambda \in L_\delta : q_1 \xrightarrow{\sigma_1' \cdot \lambda} q \wedge (\sigma_1 \cdot \sigma_2) \in (\sigma_1' \cdot \lambda)[r]_{inc}$
$\Rightarrow$ $(*$ Logical reasoning: $\sigma = \sigma_1 \cdot \sigma_2$, rewrite $\sigma_1' \cdot \lambda$ as σ' $*)$
$\exists \sigma' \in L_\delta^* : q_1 \xrightarrow{\sigma'} q \wedge \sigma \in \sigma'[r]_{inc}$

□

Lemma B.4.13 Let $s \in \mathbf{LTS}(I,U), T \subseteq \mathbf{TEST}(I,U)$ be fail fast, $s, t \in T$, $t_r \in t[r]$, $(q,q') \in Q_r \backslash (\mathsf{Fail}_r \cup \mathsf{Inconclusive}_r)$ furthermore $q' \notin L_{r\delta}$ (we exclude the T_3 states from Definition 5.5.11).

$$t_r \xrightarrow{\sigma}_r (q,q') \Rightarrow \exists \sigma' \in \mathit{Straces}(s) : t \xrightarrow{\sigma'} q \wedge \sigma \in \sigma'[r]$$

□

Proof We make use of the fact that only completely refined traces have ✓ as their second state component (in the refined test case). We identify the following cases:

- $q' = ✓$. Let $(q_0, ✓)$ be the start state of t_r.

$$\begin{array}{ll} & (q_0,✓) \xrightarrow{\sigma}_r (q,✓) \\ \Rightarrow & (*\ \text{Definition 5.5.5 (we exclude } T_3)\ *) \\ & (q_0,✓) \xrightarrow{\sigma}_r (q,✓) \wedge q \notin \mathsf{Fail}_t \\ \Rightarrow & (*\ \text{Lemma B.4.11}\ *) \\ & \exists \sigma' \in L_\delta^* : \sigma \in \sigma'[r]_{rc} \wedge q_0 \xrightarrow{\sigma'}_t q \wedge q \notin \mathsf{Fail}_t \\ \Rightarrow & (*\ \text{T is fail fast}\ *) \\ & \exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r]_{rc} \wedge t \xrightarrow{\sigma'} q \end{array}$$

- $q' \neq ✓$

$$\begin{array}{ll} & (q_0,✓) \xrightarrow{\sigma}_r (q,q') \\ \Rightarrow & (*\ \text{Definition 5.5.5 (we exclude } T_3)\ *) \\ & (q_0,✓) \xrightarrow{\sigma}_r (q,q') \wedge q \notin \mathsf{Fail}_t \\ \Rightarrow & (*\ \text{Lemma B.4.12}\ *) \\ & \exists \sigma' \in L_\delta^* : \sigma \in \sigma'[r]_{inc} \wedge q_0 \xrightarrow{\sigma'} q \wedge q \notin \mathsf{Fail}_t \\ \Rightarrow & (*\ T \text{ is fail fast}\ *) \\ & \exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r]_{inc} \wedge t \xrightarrow{\sigma'} q \end{array}$$

□

Lemma B.4.14 Let $s \in \mathbf{LTS}(I,U), T \subseteq \mathbf{TEST}(I,U)$ be a fail fast, sound and conformance trace complete test suite with respect to **ioco** and s. Note that the conformance trace completeness demand is only necessary for the only if case.

$$\sigma \in \mathit{Straces}(s[r]) \Leftrightarrow \exists t_r \in T[r], q \notin \mathsf{Fail}_r : t_r \xrightarrow{\sigma}_r q$$

□

Proof

Only if: Because of the definition of trace refinement (Definition 5.3.5) and because $\mathit{Straces}(s)[r] = \mathit{Straces}(s[r])$ (Theorem 5.3.12), we identify the following cases:

- $\sigma \in \mathit{Straces}(s)[r]_{rc}$

  $\Rightarrow$ (∗ Definition 5.3.5 (trace refinement) ∗)
  $\exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r]_{rc}$
  $\Rightarrow$ (∗ T is sound and conformance trace complete ∗)
  $\exists \sigma' \in \mathit{Straces}(s), t \in T, q_1 \in Q \backslash \mathsf{Fail} : t \xrightarrow{\sigma'} q_1 \wedge \sigma \in \sigma'[r]_{rc}$
  $\Rightarrow$ (∗ Lemma B.4.5 ∗)
  $\exists \sigma' \in \mathit{Straces}(s), t \in T, q_1 \in Q \backslash \mathsf{Fail} : t \xrightarrow{\sigma'} q_1 \wedge \sigma \in \sigma'[r]_{rc}$
  $\wedge \forall \sigma'' \in \sigma'[r] \exists t_r \in t[r] : t_r \xrightarrow{\sigma''}_r (q_1, \checkmark)$
  $\Rightarrow$ (∗ Logical reasoning ∗)
  $\exists t_r \in T[r], q_1 \in Q \backslash \mathsf{Fail} : t_r \xrightarrow{\sigma}_r (q_1, \checkmark)$
  $\Rightarrow$ (∗ Definition 5.5.11 (test case refinement) ∗)
  $\exists t_r \in T[r], q \notin \mathsf{Fail}_r : t_r \xrightarrow{\sigma}_r q$

- $\sigma \in \mathit{Straces}(s)[r]_{inc}$

  $\Rightarrow$ (∗ Definition trace refinement ∗)
  $\exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r]_{inc}$
  $\Rightarrow$ (∗ T is sound and conformance trace complete ∗)
  $\exists \sigma' \in \mathit{Straces}(s), t \in T, q_1 \notin \mathsf{Fail} : t \xrightarrow{\sigma'} q_1 \wedge \sigma \in \sigma'[r]_{inc}$
  $\Rightarrow$ (∗ Lemma B.4.6 ∗)
  $\exists \sigma' \in \mathit{Straces}(s), t \in T, q_1 \notin \mathsf{Fail} : t \xrightarrow{\sigma'} q_1 \wedge \sigma \in \sigma'[r]_{inc}$
  $\wedge \forall \sigma'' \in \sigma'[r]_{inc} \exists t_r \in t[r] : t_r \xrightarrow{\sigma''}_r (q_1, q_1') \wedge q_1' \notin L_{r\delta}$
  $\Rightarrow$ (∗ Definition 5.5.11 (test case refinement: T_1, T_2, T_3) ∗)
  $\exists \sigma' \in \mathit{Straces}(s), t \in T, q_1 \notin \mathsf{Fail} : t \xrightarrow{\sigma'} q_1 \wedge \sigma \in \sigma'[r]_{inc}$
  $\wedge \forall \sigma'' \in \sigma'[r]_{inc} \exists t_r \in t[r] : t_r \xrightarrow{\sigma''}_r (q_1, q_1') \wedge (q_1, q_1') \notin \mathsf{Fail}_{t_r}$
  $\Rightarrow$ (∗ Logical reasoning ∗)
  $\exists t_r \in T[r], q \notin \mathsf{Fail}_{t_r} : t_r \xrightarrow{\sigma}_r q$

If:

$\exists t_r \in T[r], q \notin \mathsf{Fail}_r : t_r \xrightarrow{\sigma}_r q$
$\Rightarrow$ (∗ Lemma B.4.13 ∗)
$\exists \sigma' \in \mathit{Straces}(s) : \sigma \in \sigma'[r]$
$\Rightarrow$ (∗ Logical reasoning ∗)
$\sigma \in \mathit{Straces}(s)[r]$
$\Rightarrow$ (∗ Theorem 5.3.12 ∗)
$\sigma \in \mathit{Straces}(s[r])$

□

Lemma B.4.15 Let $T \subseteq \mathbf{TEST}(I, U)$ be a test suite, $t_r \in T[r]$.

$$t_r \xrightarrow{\sigma}_r \mathsf{Fail}_r \Rightarrow \exists \sigma' \in L_{r\delta}^*, x \in U_{r\delta} : \sigma = \sigma' \cdot x$$

□

Proof This is a proof by construction. In Definition 5.5.8 (test case refinement) Unknown states are changed to Pass, Fail and Inconclusive states. A state in Unknown is added in the skeleton generation phase Definition 5.5.11. In that case rule T_3 of Definition 5.5.11 adds transitions to Unknown states. This only happens via an output action.

$\square$

Lemma B.4.16 Let $\sigma \in \mathit{Straces}(s[r]), x \in U_{r\delta}, T \subseteq \mathbf{TEST}$ is a sound test suite.

$$\forall \sigma' \in (\sigma{\cdot}x)\langle r\rangle \exists t \in T : t \xrightarrow{\sigma'} \mathbf{fail} \Rightarrow \sigma{\cdot}x \notin \mathit{Straces}(s[r])$$

$\square$

Proof Proof by reductio ad absurdum. Suppose that $\sigma{\cdot}x \in \mathit{Straces}(s[r])$

$$\begin{array}{ll}
 & \sigma{\cdot}x \in \mathit{Straces}(s[r]) \\
\Rightarrow & (*\ \text{Theorem 5.3.12}\ *) \\
 & \sigma{\cdot}x \in \mathit{Straces}(s)[r] \\
\Rightarrow & (*\ \text{Definition 5.3.5}\ *) \\
 & \exists \sigma' \in \mathit{Straces}(s) : \sigma{\cdot}x \in \sigma'[r] \\
\Rightarrow & (*\ \text{Proposition 5.3.11}\ *) \\
 & \exists \sigma' \in \mathit{Straces}(s) : \sigma' \in (\sigma{\cdot}x)\langle r\rangle \\
\Rightarrow & (*\ \text{Premise}\ *) \\
 & \exists \sigma' \in \mathit{Straces}(s) : \sigma' \in (\sigma{\cdot}x)\langle r\rangle \wedge \exists t \in T : t \xrightarrow{\sigma'} \mathbf{fail} \\
\Rightarrow & (*\ T \text{ is sound}\ *) \\
 & \exists \sigma' \in \mathit{Straces}(s) : \sigma' \notin \mathit{Straces}(s) \\
\Rightarrow & (*\ \text{Logical reasoning}\ *) \\
 & \text{Contradiction}
\end{array}$$

$\square$

Proposition 5.5.13 Let $T \in \mathbf{TEST}(I, U)$ be sound, conformance trace complete and fail fast, $t_r = \langle Q_r, I_r, U_r, T_r, q_0, \mathsf{Fail}_r, \mathsf{Pass}_r\rangle \in T[r], r : L_\tau \to \mathbf{FLTS}, \sigma \in L^*_{r\delta}$.

$$t_r \xrightarrow{\sigma}_r \mathbf{fail} \Rightarrow \exists \sigma' \in \mathit{Straces}(s[r]), x \in U_{r\delta} : \sigma = \sigma'{\cdot}x \wedge x \notin \mathit{out}(s[r] \text{ }\mathbf{after}\text{ } \sigma')$$

$\square$

Proof Because of Lemma B.4.15 we know that a trace leading to fail ends with an output. To make the proof easier to read we use $\sigma = \sigma'{\cdot}x$ for some $\sigma' \in L^*_{r\delta}, x \in U_{r\delta}$.

$$\begin{array}{ll}
 & t_r \xrightarrow{\sigma'{\cdot}x}_r \mathbf{fail} \\
\Rightarrow & (*\ \text{Definition} \longrightarrow\ *) \\
 & \exists q \in Q_{t_r} : t_r \xrightarrow{\sigma'} q \xrightarrow{x} \mathbf{fail} \\
\Rightarrow & (*\ \text{Lemma B.4.14}\ *) \\
 & \exists q \in Q_{t_r} : t_r \xrightarrow{\sigma'} q \xrightarrow{x} \mathbf{fail} \wedge \sigma' \in \mathit{Straces}(s[r]) \\
\Rightarrow & (*\ \text{Definition 5.5.8 (failure state assignment)}\ *) \\
 & \exists q \in Q_{t_r} : t_r \xrightarrow{\sigma'} q \xrightarrow{x} \mathbf{fail} \wedge \sigma' \in \mathit{Straces}(s[r]) \\
 & \wedge \forall \sigma_1 \in (\sigma'{\cdot}x)\langle r\rangle, \exists t \in T : t \xrightarrow{\sigma_1} \mathbf{fail}
\end{array}$$

$\Rightarrow$ (∗ Lemma B.4.16 ∗)
$\sigma' \in \mathit{Straces}(s[r]) \wedge \sigma'{\cdot}x \notin \mathit{Straces}(s[r])$
$\Rightarrow$ (∗ Definition *Straces* ∗)
$\forall q \in Q_{s[r]} : (s[r] \overset{\sigma'}{\Longrightarrow}_r q \text{ implies } q \overset{x}{\not\Longrightarrow}_r) \wedge \sigma' \in \mathit{Straces}(s[r])$
$\Rightarrow$ (∗ Definition *out* and **after** ∗)
$x \notin \mathit{out}(s[r] \textbf{ after } \sigma') \wedge \sigma' \in \mathit{Straces}(s[r])$

□

Theorem 5.5.14 Let $t \in \textbf{TEST}(I,U), s \in \textbf{LTS}(I,U), r : L_\tau \to \textbf{FLTS}$ and let t be fail fast and conformance trace complete w.r.t. **ioco** and s.
(t is **sound** w.r.t. **ioco** and s) $\Rightarrow$ ($t[r]$ is **sound** w.r.t. **ioco** and $s[r]$)

□

Proof We immediately use the expansions of the definitions of soundness and **ioco**. We actually prove the inverse implication.
$i \overset{\sigma}{\Longrightarrow} \wedge\, t_r \overset{\sigma}{\longrightarrow}_r \textbf{fail}$
$\Rightarrow$ (∗ Proposition 5.5.13 ∗)
$\exists \sigma' \in \mathit{Straces}(s[r]), x \in U_{r\delta} : \sigma = \sigma'{\cdot}x \wedge i \overset{\sigma'\cdot x}{\Longrightarrow}$
$\wedge\, x \notin \mathit{out}(s[r] \textbf{ after } \sigma')$
$\Rightarrow$ (∗ Definition *out* and **after** ∗)
$x \in \mathit{out}(i \textbf{ after } \sigma') \wedge x \notin \mathit{out}(s[r] \textbf{ after } \sigma')$
$\Rightarrow$ (∗ Definition **ioco** ∗)
$i \;\textbf{io}\cancel{\textbf{c}}\textbf{o}\; s[r]$

□

The following proposition shows that the Fail and Inconclusive sets can indeed be taken together if a test-suite is fail-fast and conformance trace complete.
Proposition 5.5.17 Let T be complete, fail fast and conformance trace complete with respect to s and **ioco** and let $t_r \in T[r]$.

$$t_r \xrightarrow{\sigma\cdot x} \textbf{inconclusive} \Rightarrow x \notin \mathit{out}(s[r] \textbf{ after } \sigma)$$

□

Proof
$t_r \xrightarrow{\sigma\cdot x} \textbf{inconclusive}$
$\Rightarrow$ (∗ Definition 5.5.8 (verdict assignment) ∗)
$\nexists \sigma' \in (\sigma{\cdot}x)\langle r\rangle, t \in T, q \in Q_t \backslash \mathsf{Fail}_t : t \xrightarrow{\sigma'} q$
$\Rightarrow$ (∗ Logical reasoning, use that T is conformance trace complete ∗)
$\forall \sigma' \in (\sigma{\cdot}x)\langle r\rangle : (\exists t \in T, q \in \mathsf{Fail}_t : t \xrightarrow{\sigma'} q) \vee (\sigma' \notin \mathit{Straces}(s))$
$\Rightarrow$ (∗ Logical reasoning ∗)
$\forall \sigma' \in (\sigma{\cdot}x)\langle r\rangle : \sigma' \notin \mathit{Straces}(s)$
$\Rightarrow$ (∗ Lemma B.1.7 ∗)
$\sigma{\cdot}x \notin \mathit{Straces}(s)[r]$
$\Rightarrow$ (∗ Theorem 5.3.12 ∗)
$\sigma{\cdot}x \notin \mathit{Straces}(s[r])$

□

Lemma B.4.17 Let $s \in \mathbf{LTS}(I,U), \sigma \in \mathit{Straces}(s[r])$ and let T be a exhaustive and conformance trace complete test suite for s with respect to **ioco**.

$$x \notin \mathit{out}(s[r] \textbf{ after } \sigma) \Rightarrow \exists t_r \in T[r], q \in Q_{t_r} : t_r \xrightarrow{\sigma \cdot x}_r q$$

□

Proof There is always an output that is allowed after σ (to be a valid suspension trace), if no outputs are allowed then quiescence is allowed. Therefore we can write: $\exists y \in U_{r\delta} : \sigma{\cdot}y \in \mathit{Straces}(s[r])$.

$$\begin{array}{ll}
 & \sigma \in \mathit{Straces}(s[r]) \\
\Rightarrow & (*\text{ Explained above }*) \\
 & \exists y \in U_{r\delta} : \sigma{\cdot}y \in \mathit{Straces}(s[r]) \\
\Rightarrow & (*\text{ Theorem 5.3.12 }*) \\
 & \exists y \in U_{r\delta} : \sigma{\cdot}y \in \mathit{Straces}(s)[r]) \\
\Rightarrow & (*\text{ Definition trace refinement }*) \\
 & \exists \sigma' \in \mathit{Straces}(s), y \in U_{r\delta} : \sigma{\cdot}y \in \sigma'[r] \\
\Rightarrow & (*\text{ T is exhaustive and conformance trace complete }*) \\
 & \exists \sigma' \in \mathit{Straces}(s), y \in U_{r\delta} : \sigma{\cdot}y \in \sigma'[r] \wedge \exists t \in T, q \in Q_t \backslash \mathsf{Fail}_t : t \xrightarrow{\sigma'} q \\
\Rightarrow & (*\text{ Lemma B.4.7 }*) \\
 & \exists \sigma' \in \mathit{Straces}(s), y \in U_{r\delta} : \sigma{\cdot}y \in \sigma'[r] \wedge \exists t \in T, q \in Q_t \backslash \mathsf{Fail}_t : t \xrightarrow{\sigma'} q \\
 & \wedge \forall \sigma_1 \in \sigma'[r] \exists t_r \in t[r], q' \in Q_{t_r} : t_r \xrightarrow{\sigma_1}_r q' \\
\Rightarrow & (*\text{ Logical reasoning: } \sigma{\cdot}y \in \sigma'[r] \;*) \\
 & \exists t_r \in T[r], q \in Q_{t_r} : t_r \xrightarrow{\sigma \cdot y}_r q \\
\Rightarrow & (*\text{ Definition 5.5.11: } T_3 \;*) \\
 & \exists t_r \in T[r], q' \in Q_{t_r} : t_r \xrightarrow{\sigma \cdot x}_r q'
\end{array}$$

□

Proposition 5.5.16 Let $s \in \mathbf{LTS}(I,U), \sigma \in \mathit{Straces}(s[r])$ and let T be a fail fast, exhaustive and conformance trace complete test suite for s with respect to **ioco**. As T is exhaustive we take the Fail and $\mathsf{Inconclusive}$ sets of the refined test cases together (they are the Fail state set).

$$x \notin \mathit{out}(s[r] \textbf{ after } \sigma) \Rightarrow \exists t_r \in T[r], q \in (\mathsf{Fail}_{t_r} \cup \mathsf{Inconclusive}_{t_r}) : t_r \xrightarrow{\sigma \cdot x}_r q$$

□

Proof Lemma B.4.17 shows that there is a refined test case that can execute the trace $\sigma{\cdot}x$. In the proof we show that the state where the test case ends after executing $\sigma{\cdot}x$ can only be a fail state or an inconclusive state.

Suppose that $q \in Q_{t_r} \backslash (\mathsf{Fail}_{t_r} \cup \mathsf{Inconclusive}_{t_r})$. Definition 5.5.11 shows that q is a tuple. We use the state pair (q_1, q_2) to represent q.

$\exists t_r \in T[r] : t_r \xrightarrow{\sigma \cdot x}_r (q_1, q_2)$
$\Rightarrow$ $(*$ Lemma B.4.13 $*)$
$\exists \sigma' \in \mathit{Straces}(s) : t \xrightarrow{\sigma'} q_1 \wedge (\sigma \cdot x) \in \sigma'[r]$
$\Rightarrow$ $(*$ Definition trace refinement $*)$
$(\sigma \cdot x) \in \mathit{Straces}(s)[r]$
$\Rightarrow$ $(*$ Theorem 5.3.12 $*)$
$(\sigma \cdot x) \in \mathit{Straces}(s[r])$
$\Rightarrow$ $(*$ Premise: $\sigma \cdot x \notin \mathit{Straces}(s[r])$ $*)$
$(\sigma \cdot x) \in \mathit{Straces}(s[r]) \wedge (\sigma \cdot x) \notin \mathit{Straces}(s[r])$
$\Rightarrow$ $(*$ Logical reasoning $*)$
contradiction

This means that according to Definition 5.5.11 $q \in \mathsf{Fail}_{t_r}$ or $q \in \mathsf{Inconclusive}_{t_r}$.
$\square$

Theorem 5.5.18 Let $s \in \mathbf{LTS}(I, U), T \subseteq \mathbf{TEST}(I, U)$ be a fail fast and conformance trace complete test suite with respect to **ioco** and s
(T is **exhaustive** w.r.t. **ioco** and s) $\Rightarrow$ ($T[r]$ is **exhaustive** w.r.t. **ioco** and $s[r]$)
$\square$

Proof We immediately expand the definitions of exhaustiveness and **ioco**. We actually proof the inverse implication.

$i \ \mathbf{io\cancel{c}o} \ s[r]$
$\Rightarrow$ $(*$ Definition **ioco** $*)$
$\exists \sigma \in L_{r\delta}, x \in L_{r\delta} : x \in \mathit{out}(i \ \mathbf{after} \ \sigma) \wedge x \notin \mathit{out}(s[r] \ \mathbf{after} \ \sigma)$
$\Rightarrow$ $(*$ Proposition 5.5.16 $*)$
$\exists \sigma \in L_{r\delta}, x \in L_{r\delta} : x \in \mathit{out}(i \ \mathbf{after} \ \sigma)$
$\wedge \exists t_r \in T[r] : t_r \xrightarrow{\sigma \cdot x} \mathbf{fail} \vee t_r \xrightarrow{\sigma \cdot x} \mathbf{inconclusive}$
$\Rightarrow$ $(*$ Definition out and **after** $*)$
$\exists \sigma \in L_{r\delta}, x \in L_{r\delta} : i \overset{\sigma \cdot x}{\Longrightarrow}$
$\wedge \exists t_r \in T[r] : t_r \xrightarrow{\sigma \cdot x} \mathbf{fail} \vee t_r \xrightarrow{\sigma \cdot x} \mathbf{inconclusive}$
$\Rightarrow$ $(*$ We take the union of Fail and $\mathsf{Inconclusive}$ as fail states $*)$
$i \ \mathbf{pa\cancel{s}ses} \ T[r]$

From this we conclude that $T[r]$ is **exhaustive** w.r.t. **ioco** and $s[r]$.
$\square$

Appendix C

Samenvatting

Het werk in dit proefschrift vindt plaats in de traditie van formeel testen op basis van gelabelde transitie systemen (LTS). Preciezer gezegd vindt het plaats in de traditie van conformance testing, waarbij we in het bijzonder uitgaan van de **ioco** theorie van Tretmans [Tre08]. Dit betekent dat we van het gedrag van het te testen systeem een LTS model maken, dit is een model met toestanden en acties om van toestand naar toestand te komen. Het model dient als specificatie van het gewenste gedrag van het te testen systeem. Neem als voorbeeld van een LTS een koffie automaat. In de begintoestand kunnen we geld inwerpen, waarna we in een toestand komen waar we op het knopje voor koffie kunnen drukken, waarna de koffiemachine ons koffie geeft en weer teruggaat naar zijn begintoestand.

In hoofdstuk twee geven we een overzicht van formeel testen en de **ioco** theorie in het bijzonder. De **ioco** theorie beschrijft hoe van een model een verzameling testen afgeleid kan worden die het te testen systeem in de limiet (tijd en ruimte) volledig test. Daarnaast is **ioco** een zogenaamde conformance relatie; een relatie die correctheid uitdrukt tussen model en het te testen systeem. Het toepassen van de theorie staat of valt met het gemak waarmee een model te maken en aan te passen is. Dit proefschrift gaat in op twee manieren om het maken en aanpassen van modellen te ondersteunen. De eerste betreft het opbouwen van modellen uit communicerende sub-modellen. Dit is een voorbeeld van de belangrijke verdeel en heers engineering strategie: deel het probleem op in kleinere deelproblemen. De tweede manier betreft het geautomatiseerd aanpassen van modellen met behulp van de zogenaamde action-refinement techniek. Hierbij vervangen we acties uit het model door complexer gedrag, zoals een LTS-model.

In hoofdstuk drie gaan we in op het samenstellen van een model uit communicerende sub-modellen. We laten zien dat de traditionele **ioco** theorie hiervoor tekortkomingen heeft: de **ioco** theorie werkt enkel op volledig gespecificeerde modellen. Deze tekortkoming was nog niet bekend. De reden voor de problemen met het samenstellen van communicerende systemen is gelegen in het feit dat de **ioco** theorie incomplete specificatie modellen toestaat, incompleet in de zin dat niet alle invoeracties beschreven hoeven te zijn. Dit leidt tot een incorrect voorspelling van het te testen gedrag bij het componeren van communicerende systemen. Stel bijvoorbeeld dat de hierboven beschreven koffiemachine uit twee componenten bestaat, een component die het geld afhandelt en een component die de koffie zet. Bij voldoende geld geeft de geld component door middel van de actie *start* de opdracht aan de koffie component om het koffie-maak proces te starten. Stel nu dat het model van de koffie component, zoals de **ioco** theorie toestaat, niet gespecificeerd is voor de invoer *start*. Dit betekent op model niveau dat de koffie component niet kan starten en geen koffie als uitvoer zal geven. Het te testen systeem, mits correct gemaakt, zal uiteraard wel koffie geven. Dit betekent dat de **ioco** theorie een foute voorspelling doet (niet sound is), namelijk geen uitvoer van koffie, waar een correct geïmplementeerde koffiemachine wel koffie als uitvoer heeft.

We beschrijven drie aanpakken om deze tekortkoming in de **ioco** theorie te verhelpen. De eerste betreft het automatisch volledig specificeren van specificatie modellen. De tweede betreft een aanpassing in de semantiek van de **ioco** definitie. Beide hebben tekortkomingen en lossen het beschreven probleem gedeeltelijk op. Onze laatste aanpak betreft het veranderen van de semantiek van de parallelle operator om systemen mee samen te stellen. Dit is een bevredigende oplossing voor het beschreven probleem.

Hoofdstuk vier betreft een inleiding in de action-refinement theorie. Ze beschrijft de action-refinement theorie en geeft een aantal scenario's uit de praktijk waarom action-refinement behulpzaam kan zijn voor het automatisch aanpassen van specificatie-modellen. We laten zien dat de reeds ontwikkelde action-refinement theorie niet direct toepasbaar is voor testen met de **ioco** theorie. Dit komt omdat de bestaande theorieën niet uitgaan van modellen met invoer en uitvoer, zoals de **ioco** theorie gebruikt. Verder is veel onderzoek, in het bijzonder naar non-atomic action-refinement, gericht op andere modellen dan LTS-en, waardoor dit werk niet bruikbaar is. Bovendien kunnen met de bestaande theorieën **ioco** testgevallen niet zomaar worden verfijnt.

In hoofdstuk vijf gaan we in op ons action-refinement onderzoek om binnen de **ioco** theorie modellen automatisch aan te passen. Onze bijdrage richt zich op zogenaamde atomic action-refinement. Dit betekent dat we het verfijnen van gedrag dat wordt veroorzaakt door parallellisme buiten beschouwing laten. We behandelen atomic action-refinement voor transitie systemen met invoer en uitvoer, voor traces en test-cases. We tonen aan onder welke omstandigheden het verfijnen van testgevallen uit een (abstract) model gelijk is aan eerst het verfijnen van het model en vervolgens hergenereren van de testgevallen.

We sluiten af met conclusies en enkele wegen voor verder onderzoek.

Bibliography

[Abr87] S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3):225–241, 1987.

[Axi] Axini. `http://www.axini.com`.

[BAL$^+$90] E. Brinksma, R. Alderden, R. Langerak, J. v. d. Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Second Int. Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990. Also: Memorandum INF-89-45, University of Twente, The Netherlands.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BB04] L. B. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2004.

[Bed88] M. Bednarczyk. *Categories of Asynchronous Transition Systems*. PhD thesis, University of Sussex, 1988.

[Bel10] A. Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91, Volume 2*, pages 99–119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[BJK$^+$05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[Bri87] E. Brinksma. On the existence of canonical testers. Memorandum INF-87-5, University of Twente, Enschede, The Netherlands, 1987.

[BT00] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In Cassez et al. [CJRR01], pages 187–195.

[CJRR01] F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors. *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*, volume 2067 of *Lecture Notes in Computer Science*. Springer, 2001.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Stg: A symbolic test generation tool. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer, 2002.

[CvGG92] I. Czaja, R. J. van Glabbeek, and U. Goltz. Interleaving semantics and action refinement with atomic choice. In G. Rozenberg, editor, *Advances in Petri Nets: The DEMON Project*, volume 609 of *Lecture Notes in Computer Science*, pages 89–107. Springer, 1992.

[DD89] P. Darondeau and P. Degano. Causal trees. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 1989.

[DD93] P. Darondeau and P. Degano. Refinement of actions in event structures and causal trees. *Theor. Comput. Sci.*, 118(1):21–48, 1993.

[DG91] P. Degano and R. Gorrieri. Action refinement for process description languages. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, volume 520 of *Lecture Notes in Computer Science*, pages 121–130. Springer, 1991.

[Dij69] E. W. Dijkstra. Structured programming. Technical Report EWD268-0, Technological University Eindhoven, 1969. URL:http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD268.html.

[DN87] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[DNH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DNS95] R. De Nicola and R. Segala. A process algebraic view of Input/Output Automata. *Theoretical Computer Science*, 138:391–423, 1995.

[DR95] V. Diekert and G. Rozenberg, editors. *Book of Traces*. World Scientific, Singapore, 1995.

[dVT98] R. de Vries and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the* SPIN *Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2, 1998. Ecole Nationale Supérieure des Télécommunications.

[dVT01] R. de Vries and J. Tretmans. Towards formal test purposes. In E. Brinksma and T. J., editors, *FATES 2001*, volume Number NS-01-04 of *Brics Notes Series*, pages 61–76. University of Aarhus, Denmark, 2001.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. Henzinger, editors, *Computer Aided Verification CAV'96*. Lecture Notes in Computer Science 1102, Springer-Verlag, 1996.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming – Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1–2):123–146, 1997.

[GR01] R. Gorrieri and A. Rensink. Action refinement. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 16, pages 1047–1147. Elsevier, 2001.

[Hee98] L. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* Foundations of Computing Series. The MIT Press, 1988.

[Hoa85] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO89] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International Standard IS-8807. ISO, Geneve, 1989.

[ISO96] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing.* Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve, 1996.

[JJ05] C. Jard and T. Jéron. Tgv: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[JJTV99] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In *Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE XI /PSTV XVIII '99.* Kluwer Academic Publishers, 1999.

[JTo] JTorX. `http://fmt.cs.utwente.nl/redmine/wiki/jtorx/`.

[Lan90] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 87–98. North-Holland, 1990.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Principles of Distributed Computing*, pages 137–151. 6^{th} Annual ACM Symposium, 1987. Also: Technical Report MIT/LCS/TM-387, Massachusetts Institute of Technology, Cambridge, U.S.A., 1987.

[LT89] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, U.S.A., 1988.

[LY96] D. Lee and M. Yannakakis. Principles and methods for testing finite state machines. *The Proceedings of the IEEE*, August 1996.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mon90] U. Montanari. CEDISYS: Compositional distributed systems (state of the art, research goals, references). *Lecture Notes in Computer Science; Advances in Petri Nets 1989*, 424:507–524, 1990.

[Moo56] E. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153. Princeton University Press, 1956.

[Nus97] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

[PBD94] A. Petrenko, G. v. Bochmann, and R. Dssouli. Conformance relations and test derivation. In O. Rafiq, editor, *Sixth Int. Workshop on Protocol Test Systems*, number C-19 in IFIP Transactions, pages 157–178. North-Holland, 1994.

[Pet00] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In Cassez et al. [CJRR01], pages 196–205.

[Pha94] M. Phalippou. Executable testers. In O. Rafiq, editor, *Sixth Int. Workshop on Protocol Test Systems*, number C-19 in IFIP Transactions, pages 35–50. North-Holland, 1994.

[Phi87] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Aarhus, Denmark, 1981.

[Pre04] R. S. Pressman. *Software Engineering: A Practitioners Approach.* McGraw-Hill International Editions, 2004.

[PY97] A. Petrenko and N. Yevtushenko. Fault detection in embedded components. In M. Kim, S. Kang, and K. Hong, editors, *Tenth Int. Workshop on Testing of Communicating Systems*, pages 272–287. Chapman & Hall, 1997.

[PY00] A. Petrenko and N. Yevtushenko. On test derivation from partial specifications. In T. Bolognesi and D. Latella, editors, *FORTE*, volume 183 of *IFIP Conference Proceedings*, pages 85–102. Kluwer, 2000.

[PYVB96] A. Petrenko, N. Yevtushenko, and G. Von Bochman. Fault models for testing in context. In R. Gotzhein and J. Bredereke, editors, *FORTE*, volume 69 of *IFIP Conference Proceedings*, pages 163 – 178. Kluwer, 1996.

[RW01] A. Rensink and H. Wehrheim. Process algebra with action dependencies. *Acta Informatica*, 38(3):155–234, 2001.

[Seg93] R. Segala. Quiescence, fairness, testing, and the notion of implementation. In E. Best, editor, *CONCUR'93*, pages 324–338. Lecture Notes in Computer Science 715, Springer-Verlag, 1993.

[Seg97] R. Segala. Quiescence, fairness, testing and the notion of implementation. *Information and Computation*, 138(2):194–210, 1997.

[Sud88] T. A. Sudkamp. *Languages and Machines.* Addison-Wesley Publishing Company, Inc, 1988.

[Tre94] J. Tretmans. A formal approach to conformance testing. In O. Rafiq, editor, *Sixth Int. Workshop on Protocol Test Systems*, number C-19 in IFIP Transactions, pages 257–276. North-Holland, 1994.

[Tre96a] J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 127–146. Lecture Notes in Computer Science 1055, Springer-Verlag, 1996.

[Tre96b] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] J. Tretmans. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999.

[Tre02] J. Tretmans. Testtechnieken, 2002. Course material for the course testing techniques.

[Tre08] J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[Vaa91] F. Vaandrager. On the relationship between process algebra and Input/Output Automata. In *Logic in Computer Science*, pages 387–398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.

[vdBP04] M. van der Bijl and F. Peureux. I/O-automata based testing. In Broy et al. [BJK+05], pages 173–200.

[vdBRT04] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2004.

[vdBRT05] M. van der Bijl, A. Rensink, and J. Tretmans. Action refinement in conformance testing. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June 2, 2005, Proceedings*, volume 3502 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2005.

[vdBRT07] M. van der Bijl, A. Rensink, and J. Tretmans. Atomic action refinement in conformance testing. Ctit technical report, University of Twente, August 2007.

[vGG89] R. J. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In A. Kreczmar and G. Mirkowska, editors, *MFCS*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 1989.

[vGG01] R. J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Inf.*, 37(4/5):229–327, 2001.

[vGV87] R. J. van Glabbeek and F. W. Vaandrager. Petri net models for algebraic theories of concurrency. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE (2)*, volume 259 of *Lecture Notes in Computer Science*, pages 224–242. Springer, 1987.

[vO06] M. van Osch. Hybrid input-output conformance and test generation. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2006.

[WN94] G. Winskel and M. Nielsen. *The Handbook of Logic in Computer Science*, chapter Models for Concurrency, pages 1–148. Oxford University Press, 1994.

[Wol94] A. Wolfe. Intel fixes a pentium FPU glitch. *EE Times*, 822:1, 1994.