Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-MSC-2008-16

Masterarbeit

im Studiengang "Angewandte Informatik"

Equivalence Checking of TTCN-3 Test Case Behavior

Filip Makedonski

am Institut für Informatik

Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

1. November 2008

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 20 00
Fax	+49 (5 51) 39-1 44 03
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 1. November 2008

Master's Thesis

Equivalence Checking of TTCN-3 Test Case Behavior

Filip Makedonski

November 1, 2008

Supervised by Dr. Helmut Neukirchen
Co-supervised by Prof. Dr. Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science
Georg-August-University of Göttingen, Germany

Acknowledgments

First of all, I would like to thank Dr. Helmut Neukirchen for his supervision of this thesis, his continuous advice and support. Many many thanks to Prof. Dr. Jens Grabowski, who made it all possible in the first place. Further thanks go to Benjamin Zeiss for his support, ideas and advice. Additionally, I would like to thank Thomas Rings and Steffen Herbold for proofreading.
Finally, I want to thank my family for their continuous support and encouragement throughout my studies.
Thank you all for your patience and for being there!

Abstract

Previous work has shown that refactoring can be used to improve the internal quality of TTCN-3 test cases. However, refactorings and their improper application are susceptible to introducing unintentional changes to the observable behavior of test cases. An approach to validate that refactorings and their application are indeed behavior preserving is proposed in this thesis. The original unrefactored and the refactored test cases will be checked for equivalent observable behavior. The approach is based on bisimulation, and is applied on the fly to manage the state-space explosion problem. A prototypical implementation of the proposed approach is presented briefly, along with selected examples to illustrate its application and prove its applicability.

Keywords: TTCN-3, Bisimulation, Equivalence Checking, Testing and Test Control Notation Version 3

Contents

1 Introduction **1**

2 Foundations **5**

2.1 Conventions 5
2.2 TTCN-3 6
2.2.1 Architecture 6
2.2.2 TTCN-3 Core Concepts 8
2.3 Testing of Distributed Systems 13
2.4 Refactoring 14
2.5 Equivalence Checking and Bisimulation 16

3 Requirements **19**

4 Approach **21**

4.1 Concepts 21
4.1.1 Behavior 22
4.1.2 Equivalence 26
4.2 Managing the State-Space Problem 27
4.3 Execution Environment 28
4.3.1 Instrumentation 30
4.3.2 Other Aspects 30
4.3.3 Altsteps and Default Altsteps 32
4.3.4 Handling of Timing Constraints 32
4.3.5 Traces and Tracing 34
4.3.6 Exhaustive Coverage of Test Case Behavior 35
4.3.7 Parallel Simulation and Synchronization 42
4.4 Comparison 44
4.4.1 Capturing the Test System Interface (TSI) 44
4.4.2 Incremental Comparison 46
4.4.3 Other Aspects 47
4.5 Logging 48
4.6 Approach Summary 48

5 Implementation 51
5.1 Prerequisites 51
5.2 Universal Messages 51
5.3 Simulator 55
5.4 Comparator 56
5.5 Logger 56
5.6 Overall 57

6 Case Studies 59
6.1 Extract Altstep 60
6.2 Replace Altstep with Default 69
6.3 Replace Template with Modified Template 82
6.4 Parametrize Template 85
6.5 Prefix Imported Declarations 86
6.6 Extract Function 91

7 Summary and Outlook 95

Abbreviations and Acronyms 99

Bibliography 105

1 Introduction

Systems and software testing has evolved a lot over the past few decades to meet the needs of the increasingly complex systems. Just as in regular software development, new approaches and techniques are being developed to make testing faster, easier, more flexible, more robust and more reliable, and not the least – more reusable. Modularization, object and aspect orientation, use of patterns, libraries, frameworks and special purpose languages - all have contributed to the process. The *Testing and Test Control Notation Version 3* (TTCN-3) [4, 11, 28] is a prime example. It is a test specification and implementation language designed for testing distributed systems. It evolved to feature the flexibility of modern programming languages. This thesis will deal with TTCN-3.

Unfortunately, this evolution also contributes to making testing much more complex. This issue is further affected by the increasing complexity of the systems that need testing, which in turn also plays a substantial role in the evolution of systems and software testing by raising the demands on the test systems and testing approaches. This eventually leads to the point where testing as a process becomes the subject of quality assessment and tests as software artifacts must undergo testing and validation as well.

Software complexity is a significant factor, indirectly contributing to what is referred to as "*software aging*" [21]. It can be described as software quality deterioration over time due to or as a byproduct of software evolution or the lack thereof. There are two distinct types of software aging caused either by the lack of changes to software to keep up with changing expectations or by improper changes. The latter are often realized through quick and dirty fixes, amendments and/or workarounds to achieve short term goals, often without a proper understanding of the overall software system (hence why complexity is a factor), instead of proper thorough implementations with appropriate architecture adaptations, which take long term maintainability and further development into consideration. Software aging manifests itself as a further unnecessary increase in complexity and consequently decrease understandability, which results in reduced extensibility, reusability, maintainability, and eventually efficiency. The cumulative effect of these manifestations in turn ultimately decreases software reliability in the long run. Maintenance quickly becomes a burden.

The software aging effects spread quickly, as reduced understandability and lack of motivation to fix software that has already become difficult to change, contribute even further to the expansion of the problem. Ordinary software systems, no matter how good and successful they are, eventually will be affected by the phenomenon, and test systems do not seem to fall short in this respect either.

Test systems and more specifically test cases and test suites have their own development life cycles, often adapted to keep up with new features of the *System Under Test* (SUT). As test suite complexity increases, similar issues begin to emerge in test suites as well. The lack of appropriate maintenance to ensure that the internal quality of the test suite is preserved, the application of quick fixes without a long term perspective to meet short term expectations – eventually all result in a decreased reliability. The software aging effects become apparent.

However, since test suites are used to ensure the quality of a product, a decrease in their reliability due to software aging, can eventually affect the product quality as well. Furthermore, reduced maintainability can hinder the extension and adaptation of the test suites to keep up with the evolving SUT.

In ordinary software, code restructuring is a common countermeasure. *Refactoring* [10] in particular, as a systematic approach to code restructuring, is often applied on a regular basis to attempt to slow down the effects of code degeneration. Since software test development and implementation is often based on common programming practices, and provided the flexibility of TTCN-3, it comes as no surprise that refactoring has found its way in testing as well [29, 30].

But refactoring will not go without its problems. Less-than-perfect tools for its automated application, necessity for manual application of refactorings, insufficient understanding of the system subjected to refactoring - any and all of these may result in unintentional changes to the system's behavior. This will defy the purpose of refactoring. Such effects are especially undesired in test systems, as they may have an impact on the systems being tested as well. So to make sure this does not happen, we need to check if the original behavior has been preserved after refactoring has been applied.

The lack of formal proofs for behavior preservation has been recognized as a major issue [14]. The original works [19] and [23], where refactoring was first described, suggest formal approaches based on predicate calculus, but they are deemed insufficient in [26]. Apart from that, they could hardly be used automatically. Concept analysis [24], graph rewriting [13] and algebraic refinements rules [2] have also been proposed as means for behavior preservation validation. [10] and [23] suggested that behavior preservation can be measured against specifications. Such specifications are often provided in the form of test suites. For production software, test suites can then be used as "*safety nets*" [18]. In the context of test suites, this would result in role reversal. The test suite which is the subject of refactoring will then have to be validated against the system under test before and after undergoing refactoring, as proposed by [27]. This is a notion we would be building on, in part. As simple as it sounds though, nearly genius in its simplicity, it is not sufficient. There is no guarantee that all paths of the test suite will be covered. In fact, only one path in a test case is usually executed. It could be sufficient perhaps, if we could make sure that all paths are covered, in a systematic way. This is another thought we will be building on. Neukirchen [18] and later Zeiss [29, 30] ultimately suggest that *bisimulation* [1, 15, 16, 17, 20] could be the way to check the original and refactored test case behaviors for equivalence. This is where this thesis comes into place.

The goals of this thesis are investigate the applicability of bisimulation for the equivalence checking of TTCN-3 test cases and develop an approach based on it for this very purpose. Further on, a prototypical implementation will be provided as a basis for future developments and a few field tests will be performed to prove the concept and its applicability.

This thesis is structured as follows: Chapter 2 will outline the foundations on which this thesis is based, as well as provide some background information and further references to key concepts needed to understand the following chapters. In Chapter 3, we will define our specific tasks and the scope of this thesis. Our approach and its implementation will be presented in detail in Chapters 4 and 5 respectively. Chapter 6 includes a number of case studies to illustrate the application of the approach. Finally, in Chapter 7, we conclude this thesis with a short summary and outlook.

2 Foundations

In this chapter, we will cover the foundations upon which our approach will be built. First, some conventions and definitions will be introduced for consistency. Then, we will have a brief look at TTCN-3 – the underlying technology in this thesis, focusing mostly on features and particularities that are relevant to our work. Next, we will take a brief look at testing practices and issues in distributed computing, followed by a brief introduction to refactoring in general and of TTCN-3 test cases in particular. Since our approach is based on bisimulation, we will provide some basic information on the topic as well, at the end of this chapter.

2.1 Conventions

Although the reader should be familiar with the notions of a test system, system under test, test case and test suite, here is a brief overview:

- A *test system* is the test environment used to test a target system.
- A *System Under Test* (SUT) on the other hand, is the aforementioned target system being tested for correctness or other properties (performance, reliability, etc.). It is an abstract notion that can refer to a software system, a hardware device, a bundle of these or almost anything that can be referred to as a '*system*'.
- A *test case*, in general, is a set of conditions and/or variables associated with a specific behavior of a system (usually a use case) used to determine whether particular properties or requirements are satisfied in the provided context. A *TTCN-3 test case* is a formal definition of a test system behavior according to the TTCN-3 standards specification.
- A *test suite* is a collection of test cases with similar behaviors and goals.

As we will deal exclusively with TTCN-3 test cases and test suites, all uses of the more generic terms "*test cases*" and "*test suites*" in this work will therefore refer to TTCN-3 test cases and TTCN-3 test suites respectively.

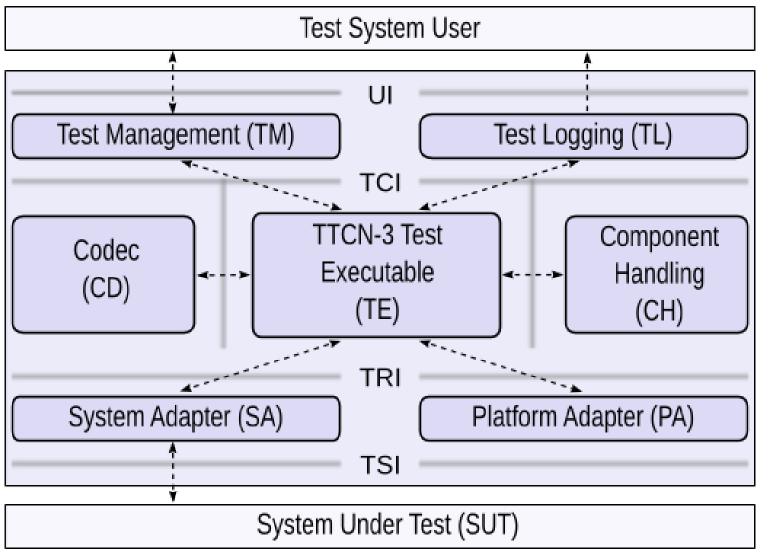

Figure 2.1: Standard TTCN-3 test system architecture

2.2 TTCN-3

The *Testing and Test Control Notation Version 3* (TTCN-3) is an internationally standardized testing language. It is mostly used for the specification and implementation of black-box tests on the system level, but can also be applied at unit and integration levels. Furthermore, it can be used for both functional and non-functional testing. It is particularly suitable for reactive systems and can also be easily used for the realization of concurrent and distributed tests.

While initially designed for the testing needs of the telecommunications industry, meanwhile it is gaining more and more recognition in various other domains, such as computer networking services and protocols, automotive systems, software systems, medical systems, GRID applications, and embedded systems.

Although this work presupposes a good knowledge of TTCN-3, we will introduce the TTCN-3 basics needed for the understanding of this thesis. Further information on TTCN-3 can be found in [4], [11], and [28].

2.2.1 Architecture

TTCN-3 test systems feature a modular architecture, which contributes significantly to their flexibility. A TTCN-3 test system consists of several entities which interact with each other during test execution. Figure 2.1 presents us with the standard TTCN-3 architecture.

At the core of the test system is the Test Executable (TE), which runs the TTCN-3 code describing the test system behavior. It can be further subdivided into generated (or compiled) code, which is the test system behavior specified on the TTCN-3 level, but in executable form, and runtime system, which implements the operational semantics of TTCN-3 [5].

The test system user needs an interface to be able to control the executable and get feedback on its operation. The functionalities available to the test system user are specified in the standardized Test Control Interface (TCI)[7] and implemented within the Test Management (TM) and Test Logging (TL) entities respectively. Their implementations present some functionality of the TCI to the user through the specific user interface.

More importantly, the test executable needs an interface to the system under test to be able to interact with the SUT and to fulfill its purpose. The set of operations available to the TE for communication with the system under test are defined in the standardized Test Runtime Interface (TRI) [6] and implemented within the system adapter. Furthermore, the TRI specifies other operational concepts such as timers and external functions which are implemented within the platform adapter.

Additionally, encoding and decoding functionalities are specified in the TCI and implemented through the Codec (CD) functional entity to provide transformation of TTCN-3 data to and from SUT-native data formats.

Finally, Component Handling (CH) is also specified in the TCI and implemented within the respective entity. The CH entity implements the creation and handling of the test components within the deployed test system.

Fortunately, most of these entities are provided by tool vendors and test developers are only concerned with aspects specific to the target SUT - CD and System Adapter (SA) and Platform Adapter (PA), base versions of which are also often provided.

To sum up, to have an operational TTCN-3 test system, one needs:

- A TTCN-3 test case that defines the behavior of the test system.
- A TTCN-3 tool – a test execution environment with a base implementation of the CH and a compiler or interpreter to turn the TTCN-3 code into executable code and run the test case.
- Test (execution) control and test logging - the “user interfaces” providing control over the test system and feedback on its operation (provided by the TTCN-3 tool usually).
- Codec - encoders and decoders to transform the outgoing TTCN-3 data into data suitable for the SUT and the other way around for incoming data respectively (base implementation also usually provided by the TTCN-3 tool).
- An SUT adapter (SA) - implements the base means of communication with the SUT.

- A PA - implements the time model and eventually external functions (again, base implementation is usually provided by the TTCN-3 tool).

2.2.2 TTCN-3 Core Concepts

In the following, we will have a look at a few key concepts specific to TTCN-3, that need to be taken into consideration in our approach.

Modules

Modules are the basic containers of TTCN-3 code. A module consists of an optional definition part and an optional control part. A simple comparison with the C++ programming language, for example, would map TTCN-3 modules to C++ classes and TTCN-3 control parts to C++ *main* methods respectively. Definitions for everything – from data types and templates to ports, components and their behavior find their place in the definition part. The control part is basically the active part. Its main purpose is to define the order of and the conditions for the execution of test cases.

Data Types and Subtyping

TTCN-3 features a rich and extensible type system. Apart from the large number of built-in data types, it provides means to create subtypes by restricting the value ranges of existing data types. Additionally, structured and list data structures can be constructed by combining built-in and/or user-defined data types and structures. The flexible type system is one of the enabling factors for the usage of TTCN-3 in a wide range of application domains, allowing the close modelling of application data from these domains.

Templates

A *template* in TTCN-3 is a data descriptor for one or more values of a specific data type. It can take different forms and serve different purposes. Templates can simply define concrete instances of a data types with fixed values. These values can be modified and result in new templates, or be assigned dynamically with the help of template parameters. Furthermore, templates can be instantiated inline for single use purposes. But as the name suggests, there is more to templates than this. A template specification can define patterns that cover a sets of values and can thus be used to check if the values of another concrete template are in these sets of values, which is then referred to as "*matching*". This built-in matching mechanism plays a key role in TTCN-3.

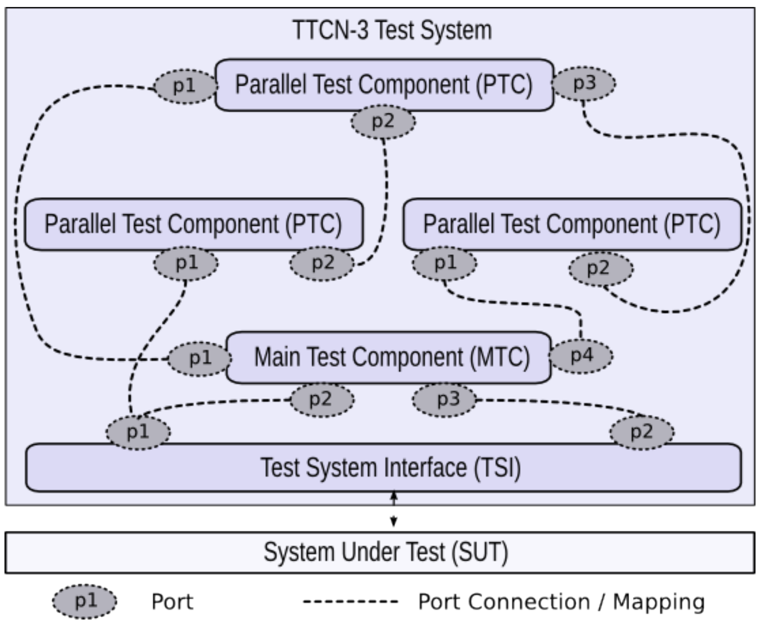
TTCN-3 Test System
Parallel Test Component (PTC)
p1
p2
p3
Parallel Test Component (PTC)
p1
p2
Parallel Test Component (PTC)
p1
p2
Main Test Component (MTC)
p1
p2
p3
p4
p1
p2
Test System Interface (TSI)
System Under Test (SUT)
p1
Port
Port Connection / Mapping

Figure 2.2: A sample TTCN-3 test configuration

Components and Ports

Test *components* act as entities that can exhibit some form of test behavior. There are two roles that test components can assume – the *Main Test Component* (MTC), on which test cases are run, and *Parallel Test Components* (PTCs), on which concurrent test behaviors can be executed. The MTC is always present if a test case is run. PTCs can be instantiated at runtime by the MTC or other PTCs. Furthermore, the abstract TSI is also specified as a component. Components can have local ports, constants, variables, and timers.

Ports define the connection points components use to communicate with each other and with the SUT. Ports are modeled as infinite First-In-First-Out (FIFO) queues in the receive direction. A port definition has a type of communication (message- or procedure- based) and types of messages that can pass through a port in a given direction (**in**, **out** or **inout**). We will be concerned only with message-based communication in this thesis, but the concepts can be translated to procedure-based communication as well.

For communications to actually take place, the ports need to be *connected* (for internal communications between test components) or *mapped* (for communications with the SUT). Port connections and mappings occur at runtime and can also change during execution. Component instances, ports, and their interconnection comprise the test system configuration. Figure 2.2 shows a sample test configuration.

Communication Basics

Message-based communication in TTCN-3 takes the form of basic *send* and *receive* operations on ports. Send operations are *asynchronous* and define the proactive behavior of the test system, whereas receive operations define the reactive behavior of the test system. There are a few peculiarities concerning receive operations:

- Receive operations are generally called with a template describing one or more specific values. A receive operation checks if the message on top of the message queue of the port on which it was called matches that template and if it does, removes it from the queue. A receive operation call without a template simply matches any message of any type and removes it from the message queue of the port it was called on.
- Receive operations are *blocking* operations. A standalone[1] receive operation call will block the currently executing behavior of the test component on whose port it was called, until there is a message on top of the *in* queue of that port that matches the template the receive operation was called with.

Timers

Timers are used to represent timing properties in TTCN-3 and have versatile functions. Timers are associated to test components. Timers can be started with arbitrary durations, stopped, but most importantly, timers can be used to guard against inactivity of the SUT (or other test components) with the help of the *timeout* operation. When a timeout operation is called on a running timer, the behavior of the test component, on which it was called, is blocked, until the running duration of the timer expires. Additionally, timers can be used to measure time, in performance measurements, for example.

Alt Statements, Snapshot Semantics and Altsteps

Alt statements are one of the most peculiar aspects of TTCN-3 test behavior. An alt statement is also one of the most natural constructs when it comes to interaction. When we interact with a system, generally, one of a few things can happen:

- It might do what we expect it to do, for example, provide correct or incorrect output for a given input.
- It might do something unexpected – it might exhibit exceptional behavior by providing output that has not been explicitly specified as correct or incorrect.
- It might just do nothing and stay inactive, until we eventually lose interest.

[1]See “Alt statements”

The reactions to such alternatives can be specified by means of an alt statement. Since both receive and timeout operations are blocking operations, they can be used to specify alternative behavior, where whichever event occurs first – an expected message is received or a running timer expires, determines the subsequent behavior of the test component. Conventional programming languages could express similar behavior through a multitude of *if / else if* statements, which are available in TTCN-3 as well. However, the semantics of an alt statement are different.

An *if / else if* construct is evaluated, and if none of the alternative branches is selected, the execution proceeds executing the last *else* statement without a further condition, if there is one, or skips the whole block altogether, if none of the conditions evaluates to true. In contrast, when an alt statement is evaluated, the execution cannot proceed, until one of the alternative branches is selected. Alt statements are thus also blocking statements. An alt statement containing only a single alternative blocking statement can be reduced to the single blocking statement without the enclosing alt statement, preserving the semantics, therefore, the opposite is also true – all standalone blocking statements can be implicitly enclosed in an alt statement, preserving the semantics. This is indeed how standalone blocking statements are treated in TTCN-3.

The alternatives in an alt statement are evaluated in a top-down order. Furthermore, to avoid race conditions, when an alt statement is encountered in the behavior specification of a test component, the state of the test component is frozen, until all branches of the alt statement are evaluated. "Frozen" means here that during evaluation of the alternatives, all the information that is relevant to their evaluation cannot change, e.g. no other events can take place and no variables can change. The state of the test component is frozen before the evaluation starts, which is referred to as a "*snapshot*" of the state of a component. Evaluation of the branches then takes place in a top-down order. If no alternative can be chosen, then a new snapshot is taken and re-evaluation takes place. This is referred to as "*snapshot semantics*". Additionally, alternative branches can optionally be guarded using boolean expressions, which may render them unavailable, if the guarding expression evaluates to false.

Often alternatives in different alt statements may be identical (timeouts, failing conditions). TTCN-3 offers the concept of *altsteps*, which are basically extracted alternatives, that can be reused. Altsteps can then be called within an alt statement, in which case, the alternatives that were extracted, are evaluated as if they were explicitly specified in the alt statement. Altsteps can also be used standalone (in which case they are implicitly surrounded by an alt statement), or be used as activated *default altsteps.*

Altsteps, used as activated defaults, are means to further reduce code duplication. Once activated, they are evaluated at the end of each alt statement, after all other explicit branches, until deactivated. Following the above reasoning, that all standalone blocking statements are implicitly enclosed in alt statements, activated default altsteps will also be evaluated after each standalone receive, timeout, and altstep statements.

Test Behavior

Test system behavior is defined by means of functions and test cases. There are two types of functions – simple or "pure" functions that do not feature communication operations[2] and component-type specific functions (declared with the **runs on** clause) that run on a specific type of test components and thus define the behavior of an instance of that test component type. Test cases are a special instance of the latter that run on the MTC. Test cases also specify the component type of the TSI.

Parameters

Parametrization plays a key role in TTCN-3. Just about anything can be passed as a parameter in TTCN-3 (timer-, port- and component- references included) and most language constructs allow parametrization (templates, functions, test cases and altsteps among others). This feature is a strong basis for reusability. Parameters declaration includes an optional passing mode, type, and name. Parameters can be passed by value or by reference. The passing mode can be same keywords as the ones specifying the direction in port definitions are used - *in*, *out* and *inout*:

- Parameters passed by value (using the **in** can be read, but any changes to their value are discarded after the end of the scope of the construct they were passed to.
- Parameters passed by reference (using the keywords **out** and **inout** retain changes to their values after the end of the scope of the construct they were passed to.

Some parameters, such as port and timer references, must always be passed by reference, whereas some contexts, such as altsteps that are activated as defaults, only allow parameters passed by value.

Summary

Thanks to the modular architecture, TTCN-3 test systems are neither tied to a particular application, nor an execution environment, a compiler or an operating system, and along with a rich type system and wide application domain provide for highly versatile and reusable software test platforms.

[2]"Pure" functions can also feature generic communication operations when supplied with ports as parameters. Such use is however not recommended.

2.3 Testing of Distributed Systems

TTCN-3 features a distributed test system model. Since the approach described in this work will basically compare test cases and check them for equivalence, we will need to cover the common pitfalls associated with testing of distributed systems.

First, we may need to clarify the terminology and classification of systems to avoid confusion. Distributed, concurrent and parallel systems all share a common background and are terms often used interchangeably. However there are small differences, particular properties associated with each type.

Naturally, all of them are parallel systems, that is, systems with components running simultaneously. Thus all of them can have the issues associated with parallel systems, such as race conditions, synchronization, or increased complexity. Testing such systems can further introduce probe effects and interference which may affect already present issues.

Concurrent systems, in relation to parallel systems, focus more on the interaction between simultaneously running components and thus testing of concurrent systems focuses on possible errors resulting from interleaving. Interleaving is the arbitrary execution of atomic statements from different parallel process streams. An atomic statement can be defined differently depending on context and purpose. A process stream is a sequence of atomic statements that defines the behavior of a process.

Distributed systems are a type of concurrent systems, that run on separate computers communicating over a network, often involving heterogeneous environments. Testing such systems has to focus additionally on interoperability and network and resource reliability.

Conventional testing of distributed systems is therefore difficult. It relies on running a program on the system with selected inputs in order to detect a fault. Due to interleaving however, the overall system may behave differently and even produce different output over repeated runs with the same input. This renders the system behavior nondeterministic.

Tai and Carver [25] identify the following problems in testing of distributed systems, due to nondeterminism:

Problem 1: Due to the nondeterministic behavior of a distributed system, a single execution of a given program with a given input may be insufficient to determine the correctness of the program with that specific input.

Problem 2: When analyzing an erroneous execution of a nondeterministic distributed program with a given input, there is no guarantee when and whether the error can be reproduced by repeated executions with the specific input.

Problem 3: After supposedly correcting the fault in the nondeterministic distributed program (for the given input), one or more successful executions with that specific input do not imply that the fault has been corrected for that input.

Relating these problems to TTCN-3 test systems and the context of this thesis in particular, we have similar issues arising from nondeterministic behavior:

Issue 1: If the test system (or more specifically the test case running on the test system) has nondeterministic behavior, it cannot determine whether the SUT reacted correctly with the output provided from the SUT and passed the test or not, therefore it will be of not desirable in the general context

Issue 2: If we are analyzing the behavior of the nondeterministic test system, it may not react consistently to an SUT that exhibits identical behavior over repeated test executions

Issue 3: If the nondeterministic test case was modified aiming to preserve its behavior by application of refactorings, we will not be able to prove that the behavior was indeed preserved

Thus, in our context, if nondeterminism is present, the same test system running the same test case against the same SUT cannot be determined equivalent to itself in terms of behavior. Nondeterministic behavior will therefore be considered a sign of bad test case design (unless specifically desired).

2.4 Refactoring

Refactoring, as expressed by Martin Fowler [10], “is the process of changing a software system in such away that it does not alter the external behavior of the code yet improves its internal structure”. It is a systematic approach to code restructuring, consisting of the application of small changes in an established sequence, which in turn aims to minimize the chances of introducing unintentional changes to the external behavior. The cumulative effect of these small changes can improve the quality of the code significantly. Furthermore, it may even impact the original system design in a positive way.

Fowler [10] provides the following two definitions to establish the terminology:

Definition: *Refactoring (n)* - a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

Definition: *Refactor (v)* - to restructure software by applying a series of refactorings without changing its observable behavior

The definitions provide the essential description of its purpose - changes to make software easier to maintain by making it easier to understand and modify, not fix bugs, not enhance functionality, not even optimize performance. This is a very important distinction. It may however precede or accompany changes aiming any of these goals, by making the code more amenable to change. As a result, the code becomes easier to enhance or extend its functionality, easier to understand and therefore perhaps easier to identify and fix bugs in it. A combination of these benefits can ultimately make performance optimization easier to identify and apply. In the bigger picture, refactoring aims to slow down and limit software deterioration and aging, which may often result in reduction of costs on the long run as well.

Refactoring has been designed to be applied manually, however many refactorings can be applied automatically as well. There are tools [9] already available for many popular languages and some refactoring functionality is present in almost every *Integrated Development Environment* (IDE). For TTCN-3, the *TTCN-3 Refactoring and Metrics Tool* (TRex) [29] provides advanced refactoring functionalities. Other IDEs allow the application of basic refactoring as well.

Another point to stress on, is that refactoring does not and should not change the observable behavior of software. External observers (users or other software and systems interacting with the software being refactored) must not be able to distinguish between the refactored and the original version.

Phillips and Rumpe [22] made the observation that “the correctness of application of a refactoring rule is in the responsibility of the developer” and that “there is (currently) no proof system that allows to formally prove correctness - neither automatic nor interactively”. They further made the observation that not all refactorings need to be justified by formal reasoning. Provided that refactorings are applied correctly, for the rather trivial ones this may be the case. However, even when applying the most trivial of refactorings, things can go wrong and therefore all refactorings need to be validated for correctness of application. We are addressing specifically these issues in the domain of TTCN-3 test cases. We do not assume that refactorings are applied correctly. We aim to provide automated validation that they are. We will informally define “the correctness of an application of a refactoring rule” as *observable behavior preservation* or more specifically *equivalence in the observable behavior before and after the application of a refactoring rule* and thus aim at proving that the later holds under all circumstances that can be externally induced and observed. Furthermore, we aim to provide means for the validation of new refactorings, which are not yet certified as “behavior preserving”

In the context of TTCN-3, Benjamin Zeiss’ work [29, 30] provides a comprehensive basis for the application of refactorings to TTCN-3, from a tool builder’s perspective, with a refactorings catalog and a refactoring tool (TRex).

2.5 Equivalence Checking and Bisimulation

Since our approach is in its essence loosely based on bisimulation, we ought to include some basic background information on the subject.

Informally, a bisimulation relation associates two (or more) systems, that behave in the same way under all circumstances. Therefore, one could think of it as one system simulating the other and vice-versa, (by matching each other's actions). The bisimilar systems are thus indistinguishable to an external observer. This is very much what the outcome of a successful refactoring application should be as identified in Section 2.4.

On a more formal note, the behavior of the systems mentioned can usually be specified as a Labeled Transition System (LTS). An LTS is essentially an abstracted version of a Finite State Automaton (FSA)), but an LTS can also contain multiple distinct automata.

Definition: A *Labeled Transition System* is formally defined as a tuple *(S, A, T)*, where:

- S is a set of states
- A is a set of actions (or labels)
- T is a relation of labeled transitions over S x A x S, such that if s and t are states and a is an action, *(s,a,t)* is then a labeled transition from state s to state t with a label a

Labels can have different meanings in different contexts. Typically they can denote actions such as accepted input or provided output, a condition on a transition, such as expected input or other condition.

Provided that the behavior of two systems can be specified using labeled transition systems, the bisimulation relation definition can then be refined: for every state in the LTS describing the behavior of one of the systems, there must be a state in the LTS describing the behavior of the other system that offers the same (labeled) transitions and vice-versa. As an equivalence relation, it has to be reflexive, symmetric and transitive.

To refine our definition even further and make it more precise:

Definition: *Strong Bisimulation* [1] is a binary relation R over the set of states of an LTS iff whenever s_1 R s_2 and a is an action and:

- if $(s_1,a,s_1^{'})$, there is a transition $(s_2,a,s_2^{'})$, such that $s_1^{'}$ R $s_2^{'}$ (relation holds for the target states as well)
- if $(s_2,a,s_2^{'})$, there is a transition $(s_1,a,s_1^{'})$, such that $s_1^{'}$ R $s_2^{'}$ (relation is symmetric)

Notation according to [1] is then $s \sim s^{'}$ or the states s and $s^{'}$ are bisimilar iff there is a bisimulation that relates them. However, such a statement is arguably misleading given the above definition. A more intuitive understanding suggests that in the above context the states s_1 and s_2 are really to be considered bisimilar and thus the notation should be $s_1 \sim s_2$.

This is the classic and generic notion of strong bisimulation introduced by David Park [20] and popularized by Robin Milner [16]. It can however be rather confusing. We can paraphrase further it for our purposes.

The classic definition allows looking for bisimulation relations within the same LTS. For a more natural understanding of bisimulation in our specific context, we will define it for two LTSs:

Definition: *Strong Bisimulation (alternative)* a binary relation R over the sets of states of two LTSs, with s_1 being a state in the domain of states of LTS1, s_2 being a state in the domain of states of LTS2, and a_1 and a_2 being actions from the domains of actions of LTS1 and LTS2 respectively, where iff whenever $s_1\ R\ s_2$

- if $(s_1,a_1,s_1^{'})$ within LTS1, there is a transition $(s_2,a_2,s_2^{'})$ within LTS2, such that $s_1^{'}\ R\ s_2^{'}$ AND $\mathrm{a}_1 \equiv \mathrm{a}_2$ (relation holds for the target states as well AND the transition labels / actions are equivalent)
- if $(s_2,a_2,s_2^{'})$, there is a transition $(s_1,a_1,s_1^{'})$, such that $s_1^{'}\ R\ s_2^{'}$ AND $\mathrm{a}_1 \equiv \mathrm{a}_2$ (relation is symmetric)

Obviously, this definition is a little more complicated, but easier and more intuitive to understand.

Apart from strong bisimulation, there is a notion of weak bisimulation which is much more relevant to our work as it is used to denote observational equivalence. The major difference is that for weak bisimulation, internal actions are disregarded. The formal definition will then be:

Definition: *Weak Bisimulation* [1] is a binary relation R over the set of states of an LTS iff whenever $s_1\ R\ s_2$ and a is an action (possibly internal - i) and

- if $(s_1,a,s_1^{'})$, there is a transition $(s_2,i^*.a.i^*,s_2^{'})$, such that $s_1^{'}\ R\ s_2^{'}$ (relation holds for the target states as well)
- if $(s_2,a,s_2^{'})$, there is a transition $(s_1,i^*.a.i^*,s_1^{'})$, such that $s_1^{'}\ R\ s_2^{'}$ (relation is symmetric)

where $(s_1,i^*.a.i^*,s_1^{'})$ is a transition from s_1 to $s_1^{'}$ that may involve any number (possibly zero as well) of intermediate internal transitions before and after the observable transition a, such that $(s_1,i,s_1^{''})^*\ (s_1^{''},a,s_1^{'''})\ (s_1^{'''},i,s_1^{'})^*$

Two states[3] s_1 and s_2 are then observationally equivalent or weakly bisimilar, written $s_1 \approx s_1$ iff there is a weak bisimulation that relates them.

[3]Here we adopt our own notation again. In the original source the states are again referred to as s and $s^{'}$.

Traditionally, bisimulation is often used for implementation verification of communication protocols. The reasons behind this are rather trivial – protocol specifications are often provided in using FSAs, protocol implementations can also be described using FSAs, and the protocol implementation must conform to the protocol specification, thereby both must describe the same behavior, possibly at different levels of abstraction. Thus, equivalence checking is required to validate that the implementation conforms to the specification, in that the implementation provides exactly the specified behavior and nothing apart from it. It could however be used to check whether two implementations or two specifications are equivalent in behavior just as well. And this is where it will come in very handy for our purposes.

Apart from that, equivalence checking based on observable behavior rather than structure is very relevant to the goals of this work, as we aim to provide means to make sure that indeed structural changes in the form of refactorings do not affect the observable behavior.

3 Requirements

In this chapter, we will set our specific goals for this project, define the problem setting and outline the base requirements for our approach, the means we will need and the key properties they need to have.

This project was started with the specific task of validating refactored TTCN-3 test cases. Validation in this context means that we need to make sure refactorings were applied successfully. A successful application of a refactoring will be defined as one that does not change the observable behavior of the original test case. To make sure that the behavior was indeed preserved, we will need to check the observable behavior of both the original and the refactored test cases for equivalence. Before we can proceed, we need to clarify a few concepts and their use in our context.

First, we need define what is considered *observable behavior* in our context. A TTCN-3 test system exhibits its observable behavior through its communication with the SUT and through the setting of test verdicts. A TTCN-3 test case is simply a program for the test system to exhibit a specific behavior. Observable behavior is henceforth determined through the interaction model of the test case. Additionally test cases have test verdicts, which also have to be monitored, even though they are not visible to the SUT.

Then, we need to define what is considered an *equivalent* observable behavior and subsequently what is a *change* to the behavior, and what sorts of changes will we be looking for. Building on our notion of observable behavior, equivalence should naturally be assessed from the SUT's perspective. Two test cases will be considered having an equivalent behavior if they interact with the SUT in exactly the same way. In our context, interacting in the same way consists of sending the same messages and reacting to received messages in the same way. Hence changes in the behavior can be different message contents, different message order, or different communication point. More drastic changes include introduction of new messages, complete omission of messages and changed test configurations.

Once we have established our grounds, we have to specify our equivalence checking approach and how do we go about it. Given the premise so far, we have two test cases, with possibly different internal structure, that must have the same observable behavior. They must interact with the SUT they were designed for in the same way and as such be indistinguishable from the SUT's point of view. The situation looks very similar to the concept of bisimulation on informal level. If we are to represent the behavior of the test cases using LTSs then it looks even closer to the bisimulation application context. Furthermore, it is a generic approach and can be applied automatically. It is therefore only natural to adopt an approach based on bisimulation.

Once we have defined our notion of observable behavior, we will need a way to produce its representation. We have executable test cases as input, and we need a representation of their behavior from the SUT's perspective as output. We will therefore need an execution environment capable of systematically producing (and managing) the complete observable behavior by creating the proper stimuli, and generating the traces necessary for comparison. There are numerous issues that should be taken into consideration – TTCN-3 specific – timers, complex data types and template, matching and wildcards; general – issues associated with parallel behavior such as possible probe effects, interference and interleaving to name a few. Most importantly, we are facing the state-space explosion problem, which is inevitable in the context of distributed systems behavior representations. So, for our approach to be scalable, we need to find a way to manage this issue.

Provided we have the a suitable representation of the observable behavior, we can proceed with comparison. Our approach to comparison will need to capture the SUT's perspective – the Points of Control and Observation (PCOs) at the TSI. Additionally, it will to implement our approach for the management of the state-space explosion problem.

Further on, if there are discrepancies, we would want to know where and why. To aid the post-validation analysis and provide clues about the changes in behavior and estimate their location, we will need logging mechanisms.

Additionally, our approach needs to be applicable with minimal overhead and mostly automatically and as such be generic and universally applicable. The means for the separate tasks like execution, comparison and logging need to be designed for standalone operation and multipurpose application. All of this will have an impact on the design and implementation of the means and methods applied.

4 Approach

In this chapter we will have a detailed look at our approach for the equivalence checking of TTCN-3 test case behavior.

In Chapter 3 we outlined the basic requirements our approach should meet:

- Allow the execution of test cases, with a systematic coverage of all behaviors that can be externally induced and observed, in all possible contexts, and generate traces for comparison
- Compare the observable behavior of two test cases from the SUT's perspective using the generated traces
- Generate logs and traces to allow quick and easy post validation analysis and pinpointing of problem areas

Thus, our approach will conceptually consist of three main modules - one for execution, one for comparison and one for logging.

But first, we should elaborate more on a few key concepts, including our definitions of observable behavior and equivalence to make them more precise and applicable. Most importantly, we should address a major issue we are facing – the state-space explosion problem.

In Section 4.1 we will have a look at our notions of several key concepts. Section 4.2 will deal with the explosion problem. Sections 4.3–4.5 will outline the design of the separate modules necessary for our approach and their key properties. Finally, Section 4.6 will summarize the overall architecture and process workflow in our approach.

4.1 Concepts

The concepts of behavior, observable behavior, and behavior equivalence are usually bound to a specific context. What is considered observable behavior in one context, may not be relevant in another. It also depends on the perspective. Same applies to the notion of behavior equivalence. In the following, we will refine these concepts for our specific purposes in the context of this thesis.

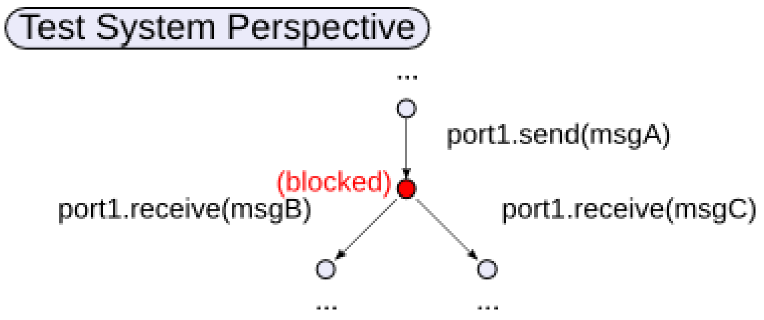

Figure 4.1: LTS representation of the sample TTCN-3 test behavior in Listing 4.1

4.1.1 Behavior

In Chapter 3 we outlined our notion of observable behavior of a test case as similar to the interaction model of a test case. It is still rather vague however. We need to refine it further, to specify which aspects of it we will be most concerned with and how we are going to represent the behavior in a formalized way. Additionally, we must consider the verdict of a test case execution, as it basically is the output of the execution.

In terms of input and output, we have a TTCN-3 test case as an input and we need to get an abstract formal representation of its behavior model that contains all the relevant information for our purposes. First, we need to define what would be considered relevant information and subsequently outline our behavior model. Then we will discuss how to produce a behavior model out of a test case.

The most relevant part of course are the messages exchanged, their contents and order. A simple interaction with the SUT for a *component1* is outlined in Listing 4.1:

```
//...
port1.send(msgA)
alt{
  [] port1.receive(msgB){/*...*/}
  [] port1.receive(msgC){/*...*/}
}
//...
```

Listing 4.1: Sample interaction behavior in TTCN-3

Using an LTS to denote this behavior segment, a possible graphical representation will look like Figure 4.1, where nodes simply stand for different states in the sequential behavior of a test component and edges denote the actions that can be taken at a given state by the test component *component1* in the test system. When a possible action is taken at a given state, the corresponding edge is traversed and the component's behavior is at the new state at the end of that edge. Additionally, red colored nodes (also tagged with "blocked"

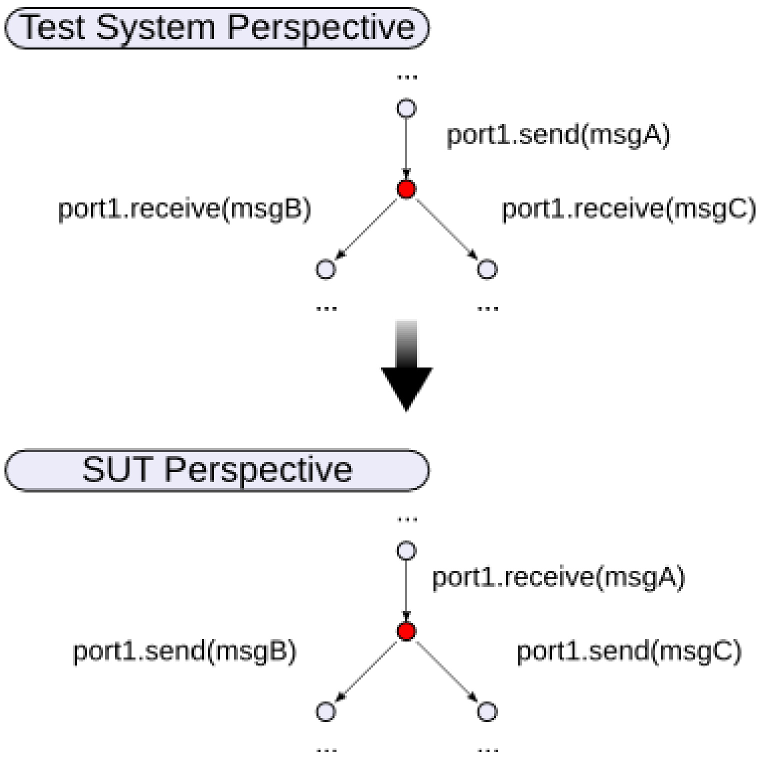

Figure 4.2: Derivation of the corresponding SUT behavior representation from the test system behavior specified in Listing 4.1

label[1]) denote states in which the behavior of the test component is blocked by a blocking statement (such as *receive*, *timeout* or *alt* statements).

On the other end of the communication channels, the SUT needs to be able to exhibit the complementary behavior, namely the SUT must be able to traverse a receive edge for every send edge traversed in the test system, a send edge for every receive edge in the test system, and have an option to traverse one of the possible edges denoting send actions when there are alternative behaviors specified in the test case. Figure 4.2 illustrates the derivation of the complementary SUT behavior.

This however is still misleading, since we have to consider the test configuration as well and specifically the port mappings to represent the complementary behavior of the SUT accurately. The ports of a test component are not transparent to the SUT. Test component ports are mapped to the TSI, which is all the SUT can see and use to communicate with the test components. More so, these mappings can change over time so we have to keep track

[1]The labels will be generally omitted to avoid clutter.

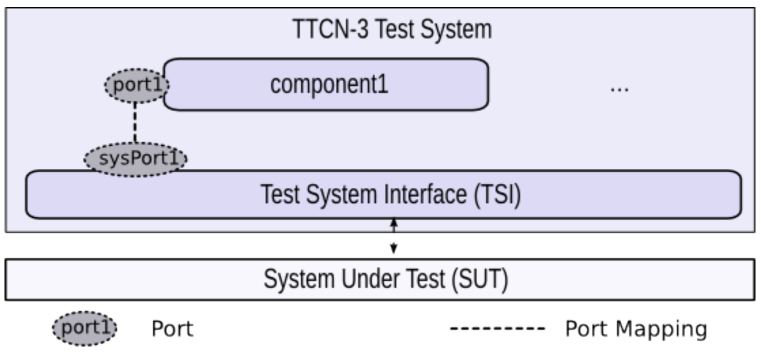

Figure 4.3: Sample test configuration for Listing 4.1 with component1.port1 mapped to system.sysPort1

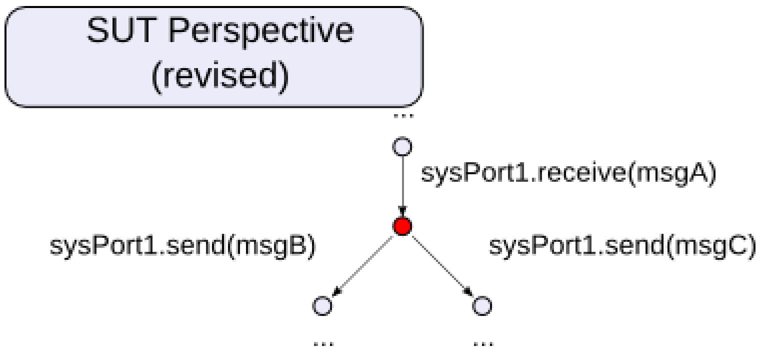

Figure 4.4: The test system behavior specified specified in Listing 4.1 from the SUT's perspective with test configuration (from Figure 4.3) taken into account

of them as well. So, given the sample test configuration in Figure 4.3, where *component1*'s *port1* is mapped to *sysPort1* on the abstract TSI, a more accurate representation of the complementary behavior in the SUT is illustrated in Figure 4.4.

Figure 4.4 may still be misleading however, since we are using the TTCN-3 notation for the behavior of the SUT as well. This may look confusing and even give the impression that the SUT's behavior is also specified in TTCN-3. To avoid such confusion, we will be using standard telecommunications notation, where *?* basically denotes a receive operation and *!* denotes a send operation. Therefore, sending a message *msgB* over a system port *sysPort1* will be expressed through *sysPort1!msgB*. Thus, the final version of our behavior representation for Listing 4.1 from the SUT's perspective using this notation is illustrated in Figure 4.5.

The necessity for an accurate representation of the SUT behavior arises from the fact, that we cannot directly control the test system and steer it into any particular behavior, let alone all possible behaviors. However, the test system can be steered by providing the

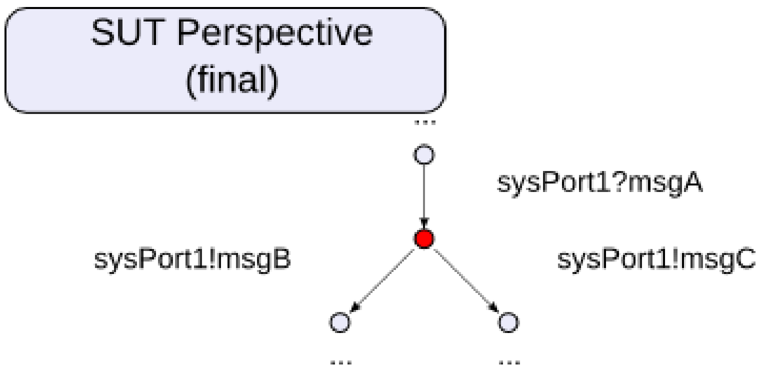

Figure 4.5: The test system behavior specified specified in Listing 4.1 from the SUT's perspective using standard telecommunications notation

proper stimuli, which come from the SUT. Therefore, by making the the SUT produce the appropriate stimuli, we can indirectly steer the test system to cover a specific behavior, or, in the general case, all behaviors.

Events

For our purposes, an action in the behavior representation (denoted by a label in the LTS) is basically any interaction event between the test system and the SUT. An event can be loosely defined as any (atomic) communication operation in the test system. An interaction event or observable event on the other hand will then be any communication operation that is visible outside the test system. A sequence of observable events describing a single execution will define a path or a possible observable behavior of the test system. The union of all such sequences will define the observable behavior of the test system. Event sequences can thus be used to model the test system behavior.

Timing

There is however more to test case behavior than simply event sequences. TTCN-3 allows the use of timeout operations on timers as options in alt statements to test for inactivity of the SUT. This is only natural, but can as well be controversial in our context. After all, timers are internal to the test components and **timeout** events are thus not directly visible to the SUT and as such not observable. But the lack of action from the SUT, no matter the causes, is in general also a type of action, only it is not a directly enforceable action and as such can also have unpredictable results. There is no way the SUT can know if it caused a *timeout* event or not and when or whether it can cause one. Since timeouts are valid and usually present options on *alt statements* and can have an impact on the test case behavior, we need to find a way to control them - we need to be able to trigger them at will and prevent their automatic triggering. Further on, if we are to check the observable behavior of two test cases for equivalence, the timing constraints they impose on the SUT should also be equivalent. Therefore, we must make timers and timeouts externally observable and controllable.

To summarize, our test case behavior representation has to capture the interaction model and the timing constraints of TTCN-3 test cases.

4.1.2 Equivalence

In Milner's words [17]: "A central question we shall try to answer is: when do two interactive systems have equivalent behavior, in the sense that we can unplug one and plug in the other - in any environment - and not tell the difference? This is a theoretical question but vitally important in practice. Until we know what constitutes similarity or difference of behavior, we cannot claim to know what 'behavior' means...". We can equally argue that until we know what behavior means, we cannot claim to know what similarity or difference of behavior means.

Opdyke [19] defines semantic equivalence generally as the property of two programs to produce the same outputs, if given the same inputs. In the context of communicating systems and test cases for such systems in particular, the situation is more complicated. We first have to identify the inputs and outputs. First, there are the incoming messages as inputs to the test case from the SUT. Additionally, there may be test case and module parameters as inputs. Then, there are the outgoing messages as outputs from the test case to the SUT and test verdicts as the main output of a test case execution. The situation is even further complicated by the fact that the incoming and outgoing messages can be related to each other.

We already established our notion of observable behavior as a union of observable event sequences, where timeouts will also be considered observable events and therefore also subjected to comparison. Additionally, we may impose timing constraints on the timing between events during comparison. However, this is beyond the scope of this thesis.

While discussing behavior we were concerned with the observable behavior of individual test components. For the purpose of this thesis, we will be concerned more with the observable behavior of the whole test system during equivalence checking. Assuming the role of the SUT, we will not be looking at individual test components, but much rather at the TSI. The TSI is basically defined by its ports, which in turn have queues for incoming and/or outgoing messages (depending on the specific test configuration). We will be using these ports and their queues as a basis for comparison, since the relevant observable events have to pass through them.

Test verdicts, as mentioned, need to be compared as well. We could monitor the dynamic changes to the verdicts, but ultimately, we are interested in the final test case verdict. Therefore, verdicts will be compared at the end of an execution only. It may be useful for the localization of issues associated with mismatching verdicts to monitor the setting of verdicts, however, we will be concerned only with the final verdicts of test cases in this thesis.

4.2 Managing the State-Space Problem

As mentioned in Chapter 3, modeling distributed system behavior using LTSs inevitably leads to the so called state-space explosion problem. When using LTSs, the behavior representation of a single test component may already become rather complicated. If we stack up a combination of the behaviors of a few test components, the situation gets much worse. If we then allow interleaving of the behaviors of all test components exhibiting parallel test behavior, it is quickly getting beyond manageable proportions. We will need to find a way to circumvent the issue, if we intend to perform exhaustive analysis.

As a first step, we will separate the representations that will be used for the different tasks – one representation for behavior management, one for behavior comparison and a combination of these for logging. For the behavior management, we will represent the behavior of each test component separately. For our ultimate goal, however, we are more concerned with the overall test system behavior and not that of the individual components. Therefore, for the behavior comparison task we will use a representation that reflects the TSI and the overall test system behavior. All of these representations will be optimized for their respective purposes.

As a second step, we will couple the processes of producing the comparison representations and comparing them to each other, all at the same time. This way, we will not need a representation of the full complexity of the system for comparison, but instead only small parts of it as necessary. This will also enable us to check test cases for equivalent behavior under execution circumstances close to the native ones. However, we will have to ensure that the full complexity is indeed covered in the process, which will require a complete representation of the observable behavior for behavior management.

The representation of the observable test case behavior can therefore still quickly grow beyond manageable proportions. We will try to prevent this from happening and perform the whole process on the fly. This means, that we can incrementally generate the representations of the observable test case behavior of both the original and the refactored test cases as needed, during their simultaneous execution and compare them to each other, discarding the covered parts. The whole process will be taking place all at the same time in parallel as illustrated in Figure 4.6.

A test case generally executes only one path of its behavior, depending on the stimuli from the SUT, and then terminates. In order to cover all the paths of the test case behavior, we have to re-execute the test case a sufficient number of times and steer the SUT into

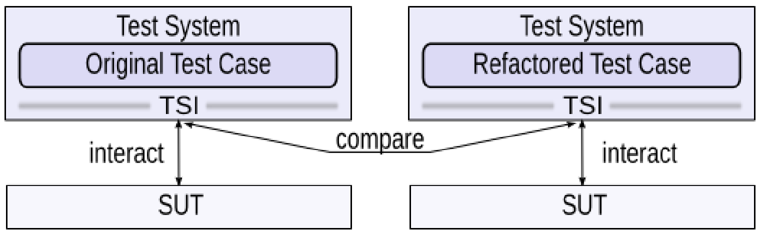

Figure 4.6: Conceptual illustration of equivalence checking on the fly

producing all the necessary stimuli at the necessary points, in order to cover all the paths of the test case behavior. We can then revise the requirements for our approach to:

- Allow simultaneous execution of two test cases with a systematic exploration of all behaviors that can be externally induced and observed, in all possible contexts, and generate incrementally traces for comparison, that can be fed to other tools on the fly.
- Compare incrementally the traces from the observable behavior of two test cases fed on the fly from the execution environment.
- Generate logs and traces to allow quick and easy post validation analysis and pin-pointing of problem areas.

In the next sections we will present in detail our solution proposal to address the revised requirements.

4.3 Execution Environment

We have identified several key properties that our execution environment should possess:

- Execute test cases and manage TTCN-3 behavior and semantics as necessary.
- Produce traces for comparison, incrementally (over a feed channel) on the fly.
- Exhaustive coverage – systematic control over test case behavior by providing all the necessary stimuli.
- Execute two test cases in parallel – the original and the refactored one, which should exhibit the same observable behavior during the individual execution - follow the same path, but be independent from each other.
- Be synchronizable.

The first key point is test case execution. A TTCN-3 test case by itself is not executable. There are basically two options – use existing general purpose execution environments or create an entirely new one specifically for our purposes. Creating a completely new execution environment for TTCN-3 is a rather complex task and will make no significant scientific contribution by itself. Apart from that, we are interested in comparing the behavior in an environment as close as possible to the native environment in which TTCN-3 test cases are used. Thus, we have opted to use existing execution environments.

But an execution environment by itself is still not sufficient to run a TTCN-3 test case. Test cases are designed to interact with an SUT which accepts stimuli from the test system and reacts to them by sending stimuli back to the test system. Thus, without a corresponding SUT, we will not get far. Here again we have two options – use the actual SUT for which the test cases are designed (provided it is available), or create a simulated SUT that somehow mimics the behavior of the actual SUT.

Using the actual SUT is closest to a real and practical setting, however, it presents us with a few key issues. Obviously, it is not a universal approach as for every test case we will need the corresponding SUT. More importantly, unless the SUT is specifically designed for testability, it will be hard to impossible to produce all the behaviors that a test case should cover, especially in a systematic way. Advanced concepts like parallel execution and timing constraints are completely out of the picture, unless the SUT design has been specifically adapted for such purposes, which may result in a significant unnecessary overhead.

A simulated SUT on the other hand, can help us circumvent these limitations. It will be designed specifically for such purposes and will provide means for exhaustive behavior coverage, parallel execution and even timing constraints. Designing a simulated SUT for each test case, to simulate a specific SUT is still not a good solution, though. It will still be too much overhead. What we are aiming at is a general purpose self-configuring simulation environment that can mimic the behavior of just about any SUT, or precisely the subset of features that a test case is designed to test. So for any test case it will automatically configure itself to exhibit the SUT properties that the test case is expecting it to, in the form of interactions - recognize the stimuli coming from the test system and send the necessary stimuli to the test system in order to steer it into all of its alternative behaviors.

We could create the SUT behavior from existing specifications, provided they exist. We could also create the SUT behavior from the test case. The latter has been even applied to create a TTCN-3 version of the SUT behavior and use a TTCN-3 test system to essentially simulate the SUT for interoperability testing of TTCN-3 execution environments [3]. It may, however, be difficult or even impossible to achieve automatically.

We are taking a slightly different approach, on par with our general “on the fly” direction. We will let our simulated SUT configure itself dynamically for exhaustive coverage during test case execution. Apparently, internal information from the test system regarding the test case behavior will be necessary. Basically, we will need to instruct our simulated SUT what stimuli it will need to send and when to do so at runtime. In other words, we will apply a look-ahead dynamic re-configuration.

Since the information we need can be easily obtained by exploiting the native TTCN-3 language constructs, it is only natural to use code instrumentation for our purposes.

4.3.1 Instrumentation

Instrumentation is basically the insertion of additional statements in the source code of a program in order to facilitate monitoring and/or control of the program's execution at runtime. In other words, we will be inserting probes into the TTCN-3 test case specification to provide inside information at runtime. We will then tie these probes to our simulated SUT, so it can interpret them and react accordingly. In the simplest form, due to the nature of TTCN-3, a useful instrumented statement can be a send statement. Listing 4.2 shows a practical example for an instrumentation:

```
...
//instrumentation begin
simPort.send(msgA);
simPort.send(expectingPort=port1);
//instrumentation end

port1.receive(msgA);
...
```

Listing 4.2: Sample instrumentation of TTCN-3 code

What we do here is basically notify over a dedicated port (*simPort*) our simulated SUT that the test system will be expecting a message *msgA* on a subsequent turn, and we further notify it that the port this message will be expected at is *port1*. So our simulated system under test will simply have to take this message, address it to the expecting port, and send it back to the test system. Figure 4.7 illustrates the interactions that take place along with the local actions at the corresponding entities before and after the interactions.

The example is oversimplified in many ways. The test system has test components, test components have ports, the ports are mapped – all of these aspects have to be addressed. Additionally, instead of plain strings, we will use a special structured type of messages, which will be called control messages, to monitor and control the execution of a test case. The instrumentation of TTCN-3 test cases is a topic of its own. Further details on the subject can be found in a separate technical report [12].

4.3.2 Other Aspects

The simulated SUT approach has various difficulties that may not be immediately apparent. In order to succeed at its tasks, our simulated SUT will additionally need to maintain a test configuration representation with port mappings and component instances to be able to steer

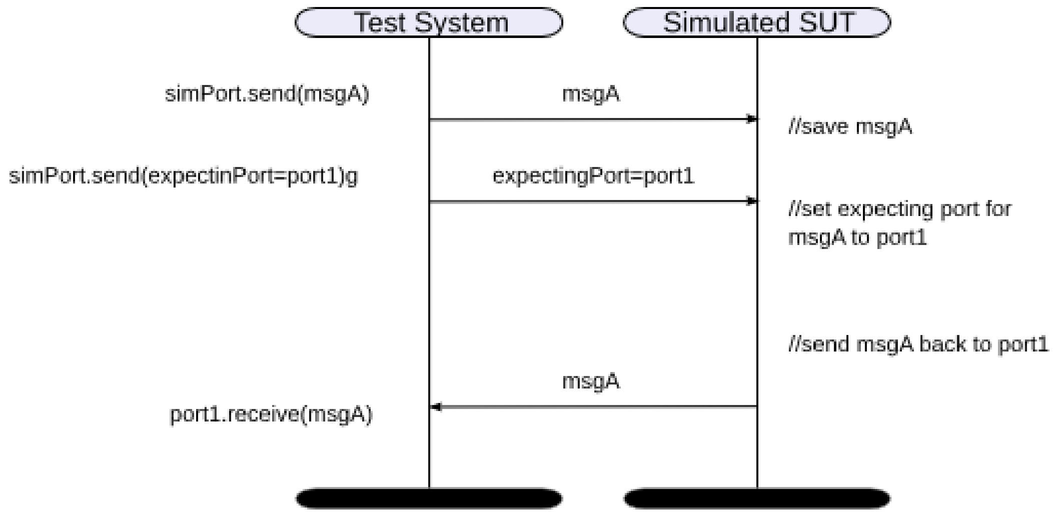

Figure 4.7: Simplified MSC illustrating the use of instrumentation at runtime

the test system sufficiently. The necessary stimuli must be dispatched through the correct communication channels, which from the SUT's perspective are the test system ports, and addressed to the correct components. To be able to dynamically resolve the addressing, our simulated SUT maintains a table with the active port mappings. Information on the port mappings can be provided via instrumentation, but is much more convenient to get from the SA, which implements the port mapping and unmapping operations. Also, we need to keep track of the component instances, and more importantly which of them are active and which are blocked expecting a stimulus. These are additional properties that we add to the behavior models.

Another important aspect is handling of matching expressions and wildcard operators. It is perhaps one of the most elusive issues to handle. We have basically two options. We can either substitute such expressions whenever possible with discrete control messages to bypass the statements with matching expressions and wildcard operators using the instrumentation. Or, using a more advanced approach, we can rely on the user to interactively supply relevant data during the instrumentation. The data will be stored in a pool so that it can be reused. During execution then, the supplied data will be used to assemble the necessary stimuli and send them to the simulated SUT, so they can be handled like regular messages. This advanced approach is covered in more detail in [12].

The **any port** construct and the **receive** without parameters construct also need special treatment. While receive without parameters could be perceived as the ultimate matching expression and treated similarly, the any port construct is a slightly different issue. We could either ask the user to specify a port during instrumentation, or we could pick a random mapped port during simulation. There could be a problem, however, when an internal communication occurs and triggers the any port branch (especially if it is rather generic). This of course could be considered a bad test case design, but it may be desired in some contexts. Its effects in such contexts can thus be ultimately considered similar to the effects of timers, therefore it is logical to apply a similar approach – substitute such statements with discrete events over which we have full control. Additionally, this way we gain a unified base for comparison – such branches can be identified, externally triggered and compared, and they will not interfere with our simulation.

4.3.3 Altsteps and Default Altsteps

Altsteps and *default altsteps* are also advanced aspects of TTCN-3 that need special handling. Altsteps are simply handled by instrumentation. When dealing with *default altsteps*, we can either handle the issue completely within the instrumentation, or, we can design a more natural handling within the simulated SUT. The latter seems more appropriate. We will approach the issue by using instrumentation to let the simulated SUT know, that there is a set of branches (the ones from the activated default), that it has to add to each and every blocking statement in the behavior specification of a test component, between the activation of an altstep as default and its deactivation. The simulated SUT will create and store a distinct representation of the branches in the altstep, associated with the test component model the altstep was activated on. Then, during test case execution, the simulated SUT will automatically attach these branches to each node that corresponds to a blocked state in the behavior representation of that test component, until the altstep is deactivated (Figure 4.8 (a)). The result will be as if the branches were explicitly attached in the conventional way (Figure 4.8 (b)).

The activated default altsteps additionally have to be put on a stack to reflect the order of their usage, should more than one altstep be activated during execution – the last one activated has to be attached first.

4.3.4 Handling of Timing Constraints

We established that timers and timeouts should be made externally observable. With the help of instrumentation, they will be substituted with discrete control messages, so we can observe and, more importantly, control them from the simulated SUT. But we still need to handle them in a special way, unlike regular messages. Naturally, timers must be handled in accordance to the TTCN-3 standards. Timer instances and their state must be maintained individually for each test component to prevent eventual problems during test case execution and comparison. The basic mechanism is relatively simple – when a timer is started, a new timer instance is created in the simulated SUT for the corresponding test component and

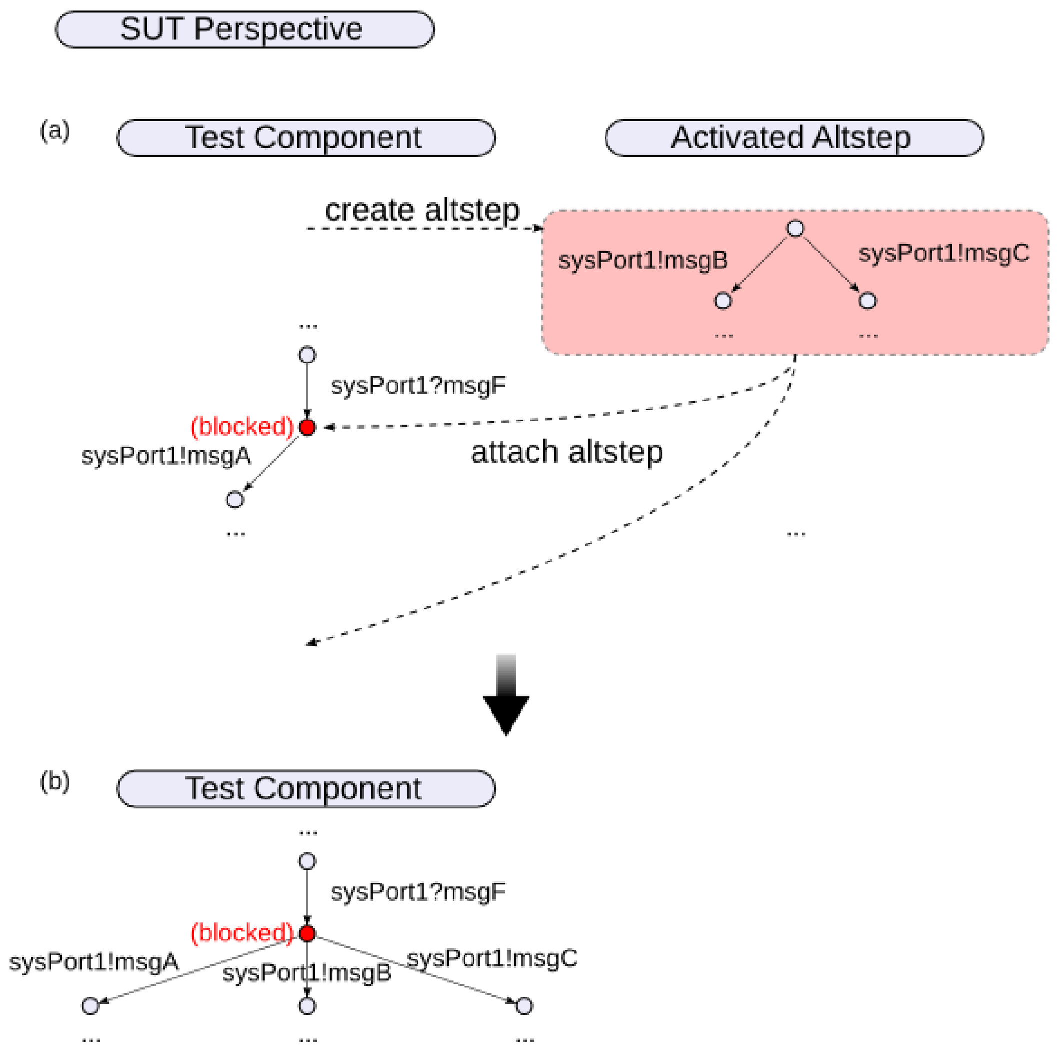

Figure 4.8: Handling of default altsteps during test case execution

its state is set to active, and when the timer is to "expire"[2], the corresponding control message will be sent to the corresponding test component, and the timer instance can then be discarded. Only existing and active timer instances can enforce timeout events by the corresponding control messages. If a *timeout* is expected from a timer that was not started or is inactive, our simulated SUT will generate an exception notification. Such problems should usually be detected by the TTCN-3 tool, but since we have substituted the original timers and with **send** and **receive** statements, there is no way the tool can know that there are inconsistencies with timers present.

4.3.5 Traces and Tracing

We have already indicated the use of internal information for the purposes of simulation. It can be perceived as tracing as well. Tracing per se however, will only be the use of information for comparison. We have limited the scope of comparison in this thesis to the externally observable behavior reflected by the messages being exchanged. Additional internal information can be exported for the purposes of comparison as well. Traces for comparison will therefore be produced generally by the *send* and *receive* operations. Their base implementation in the test system is within the SA. However, we implement the tracing functionality within our simulated SUT, which also has to implement the *send* and *receive* operations from its own perspective. Apart from trying to to be minimally intrusive to the test system, this also enables access to other information internal to the simulated SUT – behavior models in particular, which will provide necessary information to localise eventual differences in behavior. The traces themselves are of several types, most notably informative and functional. The functional traces are used for comparison. Our simulated SUT provides a way to export these functional traces over its comparison interface during execution, so that comparison can be performed simultaneously. Informative traces can be used to relate the functional traces to a specific behavior. An example for a functional trace would be a tuple *(p,d,c)*, where p is the system port a message is passing through, d is the direction in which it is going from the SUT perspective - *in* or *out* of the simulated SUT, and c is the actual message content that will be the base for comparison. An informative trace on the other hand could be the name of the test component the messages in the functional trace originated from or the event history of that test component or the whole test system up to that point in the current execution.

To have a solid basis for comparison, we first need complete coverage of the test case behavior of course.

[2]Expiration is determined not by time, but rather by selecting the branch, whose associated action will trigger the timeout control message.

4.3.6 Exhaustive Coverage of Test Case Behavior

With the help of code instrumentation and our simulated SUT, we are able to control the execution of just about any test case by sending the proper stimuli, including enforcing timeouts where necessary. Manual control however, could be tedious, and, if we seek a systematic coverage, error-prone as well. To achieve a full coverage, an automated approach for the selection of the proper stimuli is desirable.

So far, our simulated SUT can create and send the stimuli expected by the test system, but when there are multiple alternative stimuli that are expected at a given point, it cannot decide which one to create and send. Thus, whenever a stimulus is expected by the test system, our simulated SUT will provide us with the available options and allows us to manually pick one of the possible actions interactively. We have designed an algorithm to systematically make these choices until all possible options are covered. This is where the behavior models become useful.

The models are built incrementally during execution, expanding as necessary. After an execution is completed, backtracking is performed to mark the covered parts and optionally discard them for optimization.

```
testcase tc() runs on sampleComponentType{
  //...
  map(mtc:port1, system:sysPort1;
  //...
  port1.send(msgA);
  alt{
    [] port1.receive(msgB){
      port1.send(msgD);
      alt{
        [] port1.receive(msgE){
          port1.send(msgG);
          setverdict(pass);
        }
        [] port1.receive(msgF){
          setverdict(fail);
        }
      }
    }
    [] port1.receive(msgC){
      port1.send(msgH);
      setverdict(inconc);
    }
  }
  stop;
}
```

Listing 4.3: A sample test case fragment to illustrate exhaustive behavior coverage

The algorithm therefore consists of two phases – an execution phase, where execution paths are selected, and a backtracking phase, in which covered paths are marked as "*checked*". During an execution, behavior models are incrementally constructed for each separate test component. At points where alternative actions are possible, the first available is taken. The "*first available*" is the leftmost "*unchecked*" edge from the current node. Existing segments are simply being walked through and new branches are then inserted where necessary. Once a leaf node is reached at the end of an execution, backtracking is performed to mark covered segments up to the last branching node with further *unchecked* edges as *checked*. Then, during a subsequent execution, the next *unchecked* branch is taken. Given the sample test case fragment in Listing 4.3, whose complete behavior representation is shown in Figure 4.9, Figures 4.10, 4.11 and 4.12 illustrate graphically the steps taken during execution and backtracking of the three executions necessary to cover the observable behavior of that fragment. We have omitted the labels for the actions to avoid complicating the graphical representations further.

The algorithm for exhaustive behavior coverage of a single test component is outlined below:

Execution Phase – incremental behavior model building and exploration

1. Create a new empty behavior model for the test component. If a behavior model for the test component exists, use it.
2. For each message sent from the test system, add an edge with that action as a label, and a node at the end of that edge, then traverse this edge. If the edge exists, simply traverse it.
3. Where alternative actions are possible, add edges for **all** the actions and nodes at the ends of these edges and then traverse the leftmost edge. If the edges exist, simply traverse the leftmost **unchecked** edge.
4. Proceed to the end of the execution, reaching a leaf node

Backtracking Phase – model marking

1. Start at the leaf node reached at the end of the execution and proceed backwards, marking the leaf node as "*checked*". This is now the *current node*.
2. Mark the edge leading to the *previous node* in the current path as "*checked*".
3. If the *previous node* has no other outgoing edges, or if all of its outgoing edges are checked, mark it as "*checked*" and make it the *current node* (step 1), then proceed backwards, repeating steps 2 and 3. Otherwise leave the node as unchecked and stop backtracking.

Then, during the subsequent execution, segments that have already been created will be simply traversed, new segments will be inserted as necessary, and branches marked as "*checked*" will not be selected for traversal.

Additionally, to optimize and reduce the representation of the behaviors further, the backtracking algorithm can be extended to feature a reduction operation during step 3:

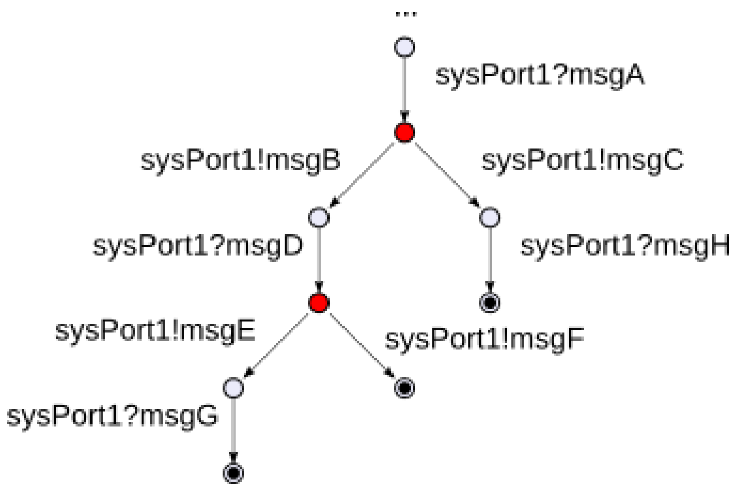

Figure 4.9: Complete behavior representation of the test case fragment in Listing 4.3

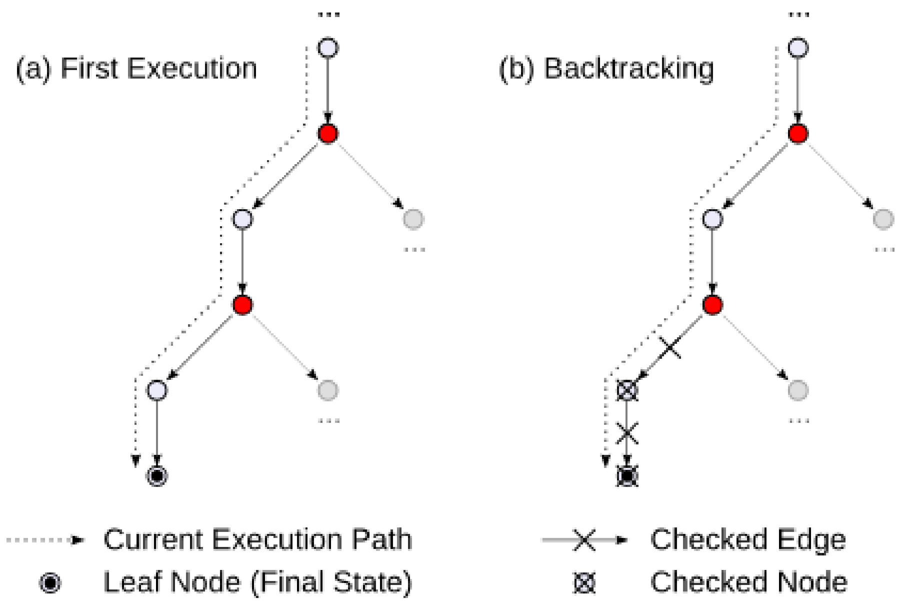

Figure 4.10: Behavior coverage during the first execution of the test case fragment in Listing 4.3 (a) and marking nodes as "checked" during the backtracking phase (b)

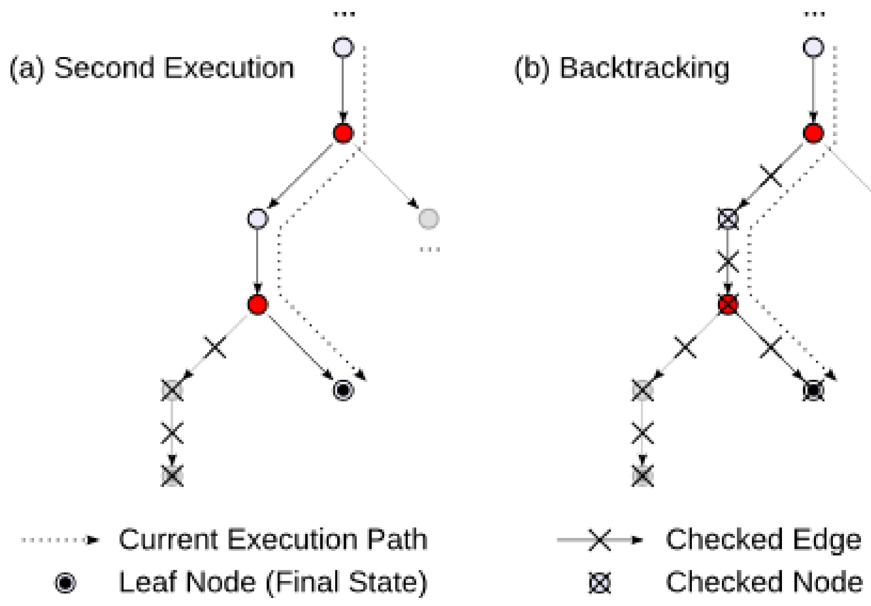

Figure 4.11: Behavior coverage during the second execution of the test case fragment in Listing 4.3 (a) and marking nodes as "checked" during the backtracking phase (b)

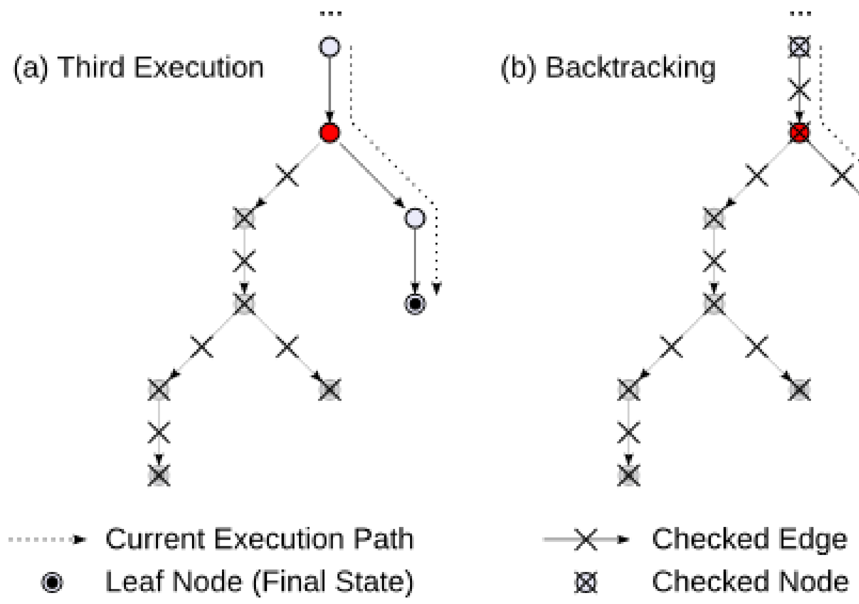

Figure 4.12: Behavior coverage during the third execution of the test case fragment in Listing 4.3 (a) and marking nodes as "checked" during the backtracking phase (b)

Backtracking Phase with Reduction - model marking and reduction

1. Start at the leaf node reached at the end of the execution and proceed backwards, marking the leaf node as "*checked*". This is now the *current node.*
2. Mark the edge leading to the *previous node* in the current path as "*checked*".
3. If the *previous node* has no other outgoing edges, or if all of its outgoing edges are checked, mark it as "*checked*", make it the *current node* (step 1) and discard its outgoing edges and the nodes attached to them, then proceed backwards, repeating steps 2 and 3. Otherwise leave the node as unchecked and stop backtracking.

With the reduction step, we aim to minimize the number of actively maintained states. The effects of such reduction can be visualized as shown in Figures 4.13 and 4.14. For the rest of this work we will be using primarily reduced representations to minimize the complexity of presentation, and thus we will assume that the backtracking algorithm with reduction was applied, unless otherwise specified. Thus, Figures 4.15 and 4.16 visualize the reduced versions of the behavior models from Figures 4.11 and 4.12 respectively.

When we have a test configuration with multiple test components, another level of complexity is added. We need to cope with multiple parallel behaviors. To achieve an exhaustive coverage, we will apply our algorithm to all the behavior models in a systematic "*token-passing*" fashion. The general principle is: *only the test component that has the token can perform backtracking.* The token will be passed on when the behavior of the test component that has the token is fully covered. So we will have all the test components, apart from the token-bearing one, taking their *first available* path. We have chosen to initially give the token to the last initialized test component to maintain a consistent and consequent order of token-passing. For this purpose, our simulated SUT maintains an absolute order of initialization. Note that passing the token in one direction is not sufficient for exhaustive coverage. If we pass it only backwards according to the initialization ordering, we will achieve full coverage of the individual test component behaviors, but not of the complete test system behavior. To cope with this issue, we need to pass the token back forward once we have selected a different path in a previous component. Given the sample behavior representations for the two test components on top of the initialization stack (the last ones to be initialized during test case execution) in Figure 4.17, Figure 4.18 illustrates visually the application of the *token-passing* algorithm during backtracking after the first three executions (the action labels have been omitted to avoid clutter). Backtracking after the remaining three executions proceeds in a similar fashion.

The token-passing mechanism outlined so far can be easily realized with the help of a stack. All the behavior models are pushed on the stack in the order of initialization. Backtracking is then performed only on the behavior model on top of the stack. Once the behavior model is completely covered, it is popped off the stack and backtracking is performed on the

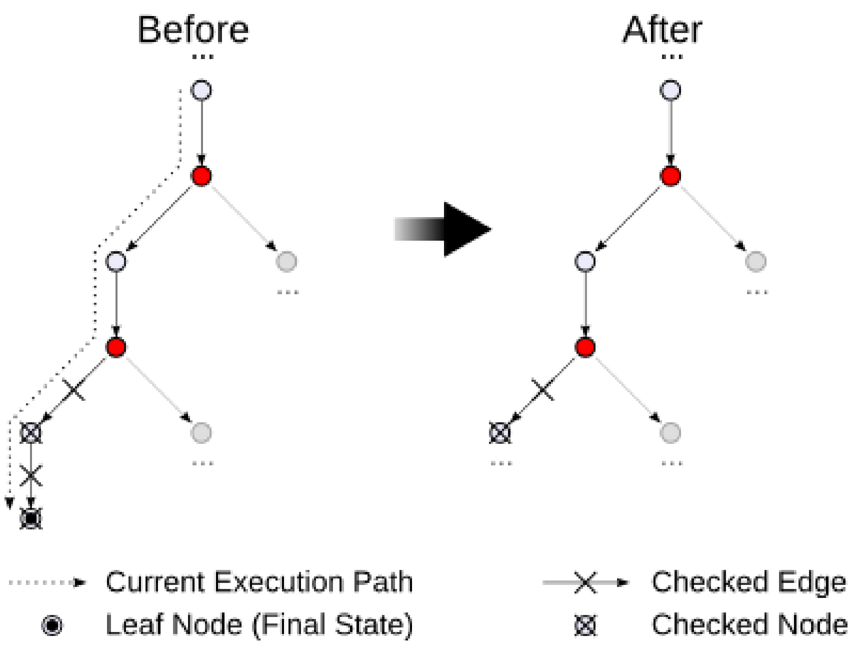

Figure 4.13: Reduction of the behavior models during backtracking after the first execution

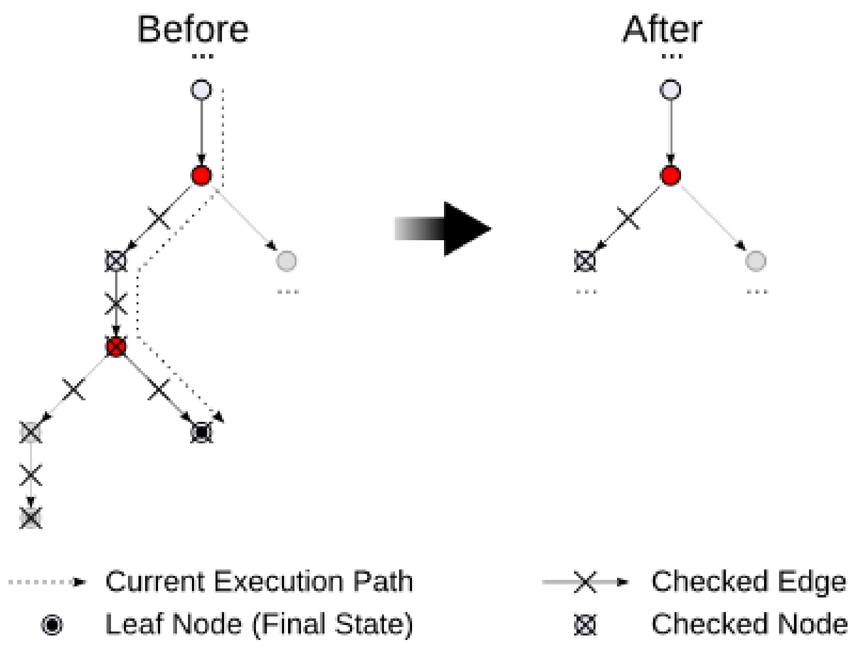

Figure 4.14: Reduction of the behavior models during backtracking after the second execution

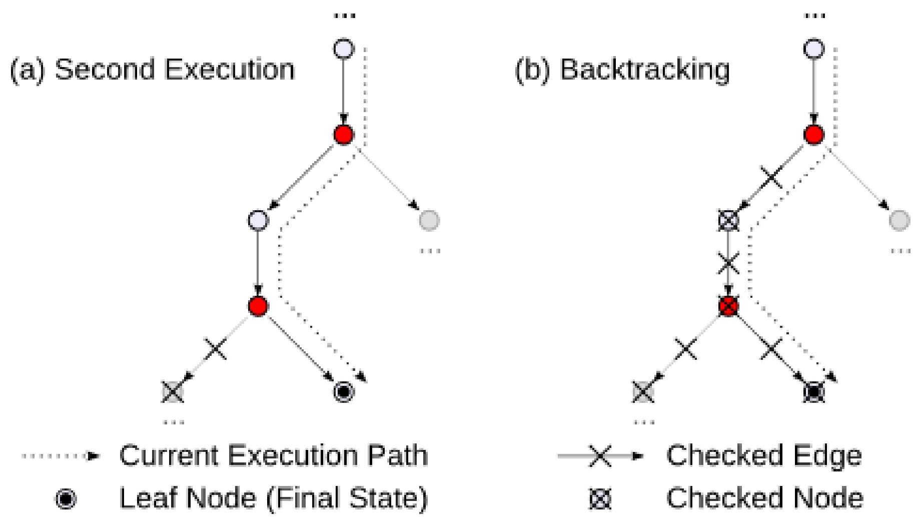

Figure 4.15: Reduced versions of the behavior models from Figure 4.11 after applying backtracking with reduction

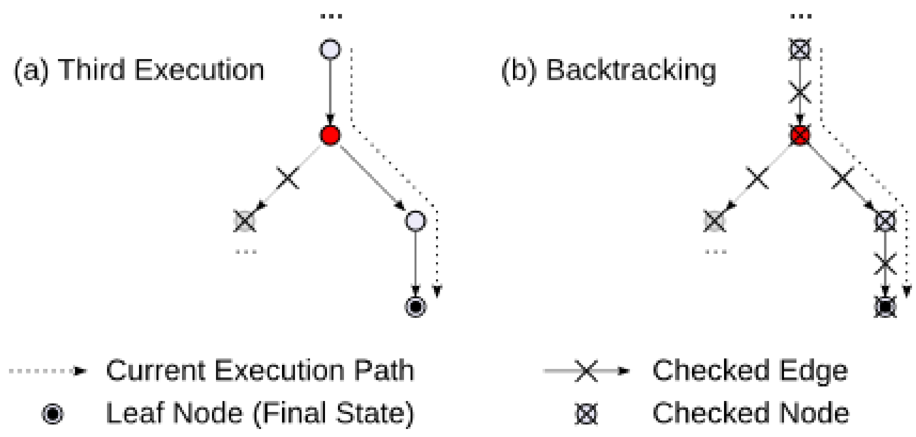

Figure 4.16: Reduced versions of the behavior models from Figure 4.12 after applying backtracking with reduction

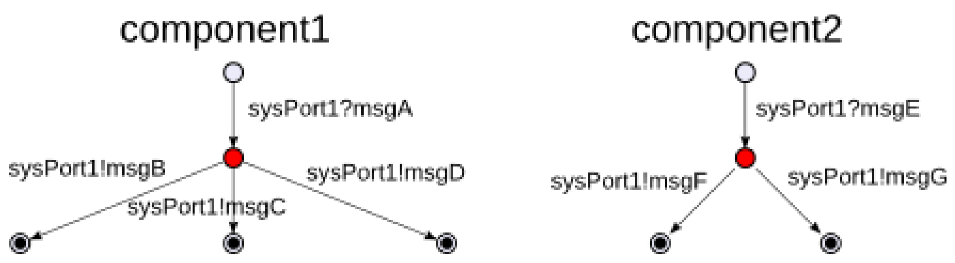

Figure 4.17: Sample full behavior representations for the two last initialized test components

behavior model below it. On a subsequent execution the behavior model for the component that was popped off the stack is reinitialized and pushed back on top of the stack. Below is the algorithm for a multi-component test configuration.

Token-Passing Backtracking - systematic backtracking for multiple test component behavior models

1. Create an empty behavior model for all newly initialized active test components and put them on a stack
2. Take the *first available* path through all test component behavior models
3. At the end of an execution, perform backtracking on the test component behavior model on top of the stack to mark its current path up to the first branching node with unchecked branches as "*checked*"
4. Repeat 2 and 3 until all paths in the test component behavior model on top of the stack are covered, then
   a) Pop the model out of the stack and discard it
   b) Perform backtracking on the test component behavior model currently on top of the stack
   c) On the subsequent execution create a new empty behavior model(s) for the test component(s) discarded and push it (them) back on the stack
5. Repeat all steps until all paths in all test component behavior models have been covered

So with the help of the algorithms and the behavior models on which they are applied, we can now automatically control the execution(s) of a test case to systematically cover all the possible behaviors. They are implemented within the simulated SUT.

4.3.7 Parallel Simulation and Synchronization

Our ultimate goal is to compare two test cases (or more accurately two versions of a test case) on the fly, therefore we need to be able to execute them simultaneously. The two test

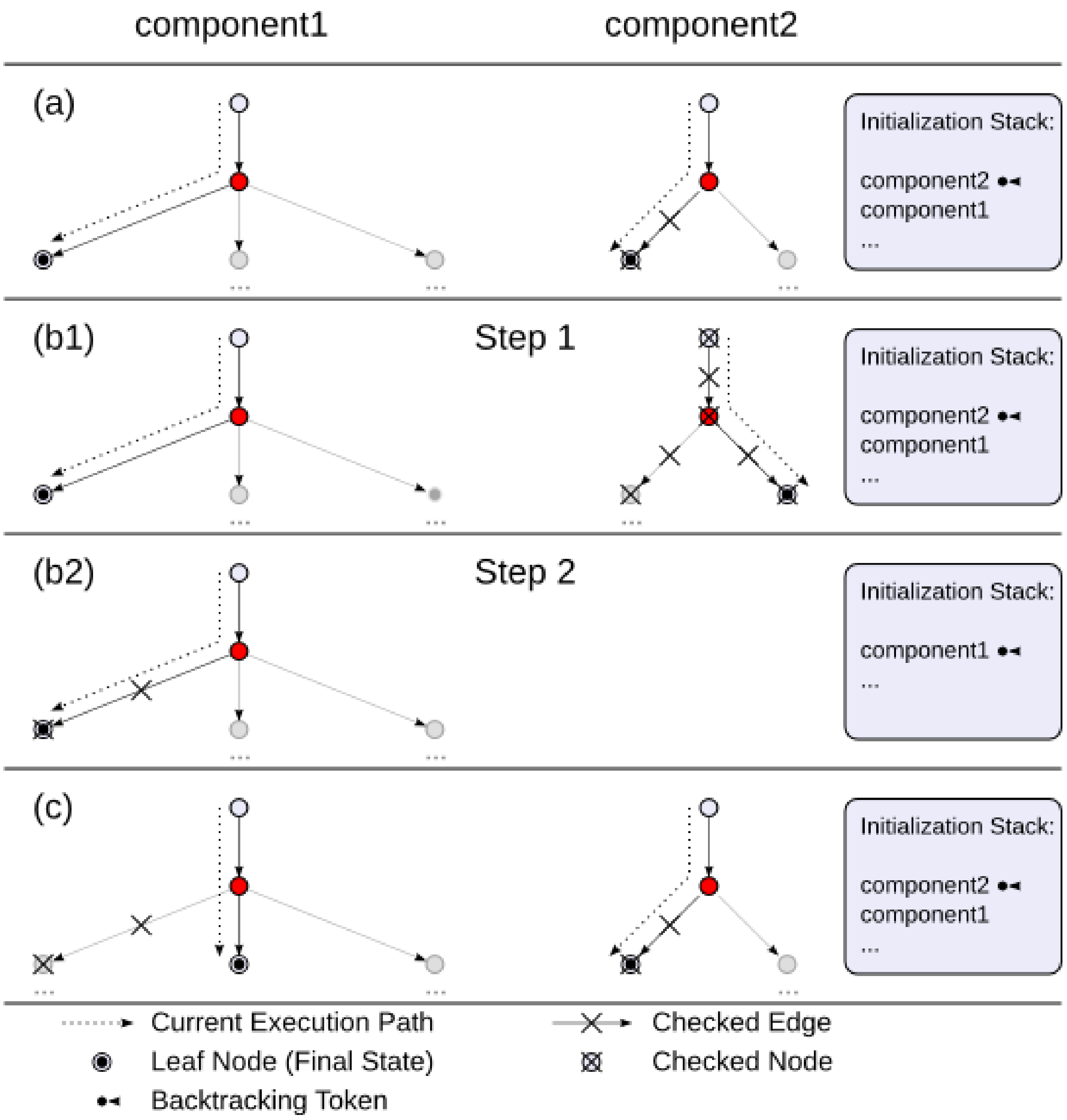

Figure 4.18: Illustration of the token-passing algorithm applied to the sample behaviors in Figure 4.17 during after the first (a), second (b1,b2) and third (c) executions

cases must be executed independently, but nevertheless provide means for synchronization (another reason to substitute timeouts with discrete events – they may severely interfere in such a context). We do not have direct control over the proactive behavior of a test system – it may send stimuli at will. We do however have control over its reactive behavior. And since *receive* operations are blocking operations, we can partially control the execution progress. *Receive* operations however, only block the behavior of the test component that reached a *receive* statement. All other active test components can still proceed with their execution. Thus, a test system can be considered in a fully blocked state only when all the active test components are blocked. This is what we will refer to as a "*stable*" test system state – where no further actions are taking place until further notice. These *stable* states will be used to synchronize the simultaneously executed test cases. Such *stable* states are essential for our approach, since they enable us to perform incremental comparison on the fly. If there are active communications while comparison is performed, it may lead to inconsistencies. Therefore, it is necessary that we provide means to control such synchronization.

Our synchronization mechanism consists of two basic steps – *block* and *release*. When a component reaches a blocking statement, the simulated SUT will be notified that it has to respond by sending an appropriate stimulus. At this point, if synchronization is enabled, the test component will enter a FIFO queue of blocked test components and wait until a "*release command*" is issued. A *release command* simply removes the lock on the test component so that an appropriate stimulus can be sent to it and it can continue its execution. A "*global release command*" is used when the test system has reached a *stable* state and basically issues *release commands* to the all the test components in the order they were blocked (using the FIFO queue). If synchronization is disabled, the *block* and *release* mechanism is bypassed and stimuli are sent immediately.

Another aspect of synchronization is the ability to allow *directed control*, by which we mean that one simulated SUT can overtake control of the other simulated SUT by selecting which stimuli it should send to the test system, whenever there are multiple options. This is an advanced concept and is best illustrated by an example. Section 6.2 illustrates how this works and deals with a situation where it is necessary.

4.4 Comparison

We have so far settled on the fact that that comparison will be performed on the fly. However, we still have to outline the basics of comparison and then elaborate on further specifics of our approach such as on the fly application.

4.4.1 Capturing the TSI

First and foremost, we stated that our comparison module should capture the SUT perspective. It should reflect the TSI, because that is what the SUT can see, not the individual

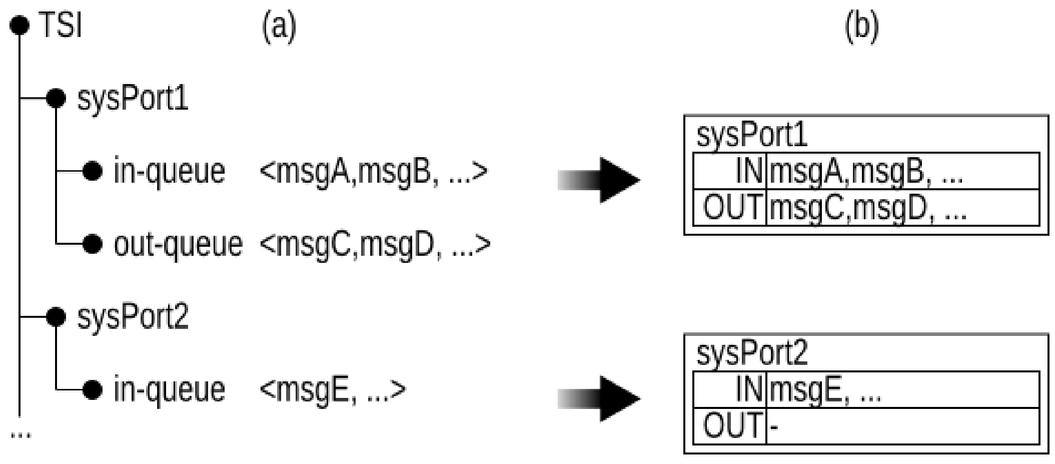

Figure 4.19: TSI conceptual hierarchy (a) and our compact representation to reflect this hierarchy (b)

behavior models for the separate test components. The TSI consists of ports - system ports, which are mapped to the ports of the individual test components. The ports have queues – input and / or output queues for the incoming and outgoing messages, depending on the individual port specifications. The queues finally contain the messages being exchanged. The messages, are thus on the lowest level. Figure 4.19 (a) outlines the conceptual hierarchy.

Our comparison module needs to reflect the above hierarchy (Figure 4.19 (b)). Furthermore, it needs to dynamically and automatically reconfigure itself, to suit the specific TSI. Comparison is performed on the lowest level (messages), with the TSI structure also taken into consideration. Given two such structures, we will be comparing the respective messages in the respective queues on the respective ports[3] (Figure 4.20).

The messages that will be compared are not the TTCN-3 messages as defined in the test specification. Instead, we will compare the encoded ready-to-be-delivered-to-the-SUT messages. The reasoning behind this is that this is exactly what a regular SUT will send and receive. It will have no idea and would not care less about TTCN-3 types and templates. It has its own data types and structures and it is the job of the TTCN-3 codecs to translate the TTCN-3 data types into the SUT specific ones (and the other way around). Furthermore, in the course of application of a refactoring, names of fields in data structures, or even whole data structures may change, which will make comparing them more difficult or even impossible. This is why we will build on the existing codecs, which are also required for the test system to be operational anyway. We rely on them to resolve the data types correctly

[3]In a situation where system ports may be renamed in the course of application of a refactoring, the ports may need to be aligned manually between the original and the refactored test case (some automated guesswork could be employed to achieve this, but so far it is considered a non-essential feature). Such situations however are probably not very desirable, as they may require further adaptation of the SA as well.

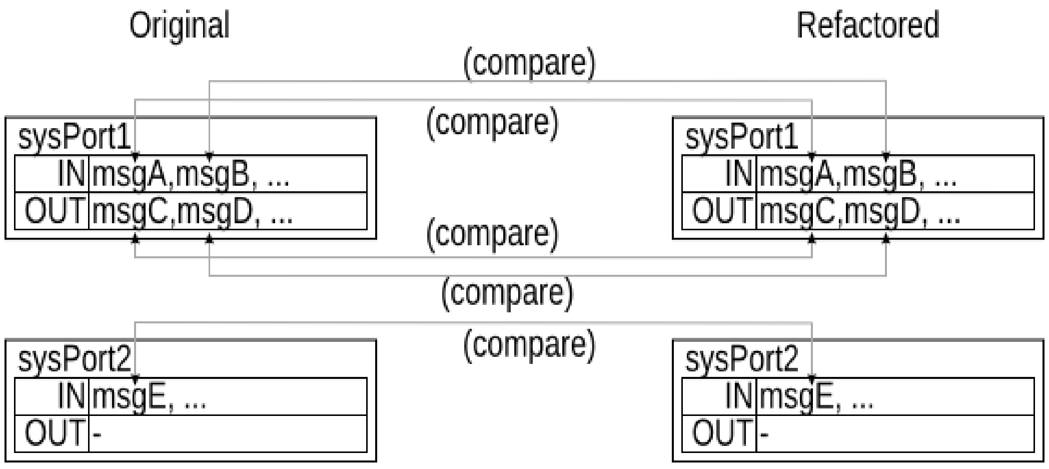

Figure 4.20: Comparison of TSI structures

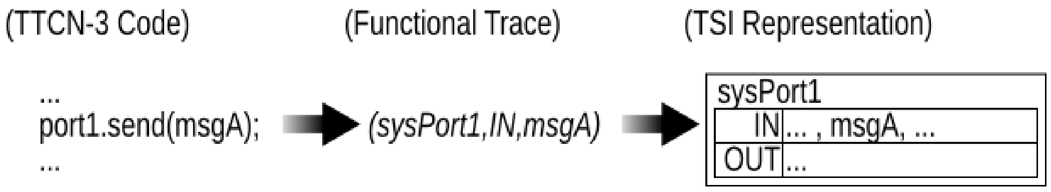

Figure 4.21: Tracing process

(as defined by the TTCN-3 standards [7]). This way we will not have to implement our own data resolution modules.

The messages for comparison will come in the form of traces, functional traces in particular, from the simulated SUT. The functional traces come as an encapsulated tuple *(p,d,c)* (or *(port,direction,contents)*), which is then decomposed and assigned to a port and a queue. Figure 4.21 illustrates the process.

4.4.2 Incremental Comparison

The comparison module needs to implement the concepts of our on the fly approach, in particular the incremental comparison and synchronization. As mentioned, we can only compare the two simultaneously running test systems whey they are in *stable* states, where no further interactions are allowed to occur. For the purposes of this thesis, such *stable* states were defined as states in which all the active test components are in a blocked state (encountered a blocking statement). At such points the comparison module can safely inspect and compare its representation of TSI - the port queues of the two test systems.

These points will be referred to as "*comparison points*". After comparison is completed, the comparison queues are emptied and the locks on the test components in the simulated SUT are released via a *global release*.

Our behavior models and exhaustive execution algorithm ensure that branch precedence in *alt* statements is preserved, meaning that a change in the order of alternative branches will be reflected in the behavior model and subsequently in the traces and therefore discovered during comparison.

Further constraints such as timing between events can also be implemented within the comparison module and associated with the necessary traces.

4.4.3 Other Aspects

Matching expressions and wildcard operators are an even bigger problem for comparison. Using the simpler solution outlined in Section 4.3.2, we need to assign the exact same discrete control events in both the original and the refactored test case, which should correspond to each other at the specific points, meaning we should use a consistent schema.

Using the more advanced solution as outlined in Section 4.3.2, we need to use a shared pool while instrumenting the original and the refactored test case and align the values where no obvious alignment is possible. Then, at runtime, the same message contents will be assembled from the pool for the specific templates in both test cases. Additionally, **any port** and **receive** without parameters branches need to be handled in a special way. Using discrete control events to substitute such branches will make comparison easier, albeit inaccurate.

A combination of both approaches may prove to be a better solution. We could still use substitution during instrumentation to provide a simple way of activating the branches or individual receive statements, but additionally we could provide the signature (template definition) of the matching expression. This way, we could analyze the signature by reducing it to the basic data types with the help of the codecs and then comparing the constraints on these basic types. Such a solution is nevertheless still on a conceptual level. Its practical applicability remains to be determined. Additionally, for the comparison of data other than well defined comparison traces, the comparison module may need an additional interface.

So far we have limited comparison to queue order and contents. In some cases it may be desirable to consider the sequential ordering of events as they occur in the separate behaviors of the individual test components. But since the SUT cannot see the individual test components directly, we have to provide partial ordering rules for the events occurring sequentially within a single test component's behavior [8]. This way we can validate that the sequence of events within sequential behavior specifications is preserved.

4.5 Logging

Should any discrepancies occur in the process so far, the comparison module will notify the logging module to register the differences and add information as to what led to it for post-analysis.

We have chosen a rather simplistic approach to logging. The logging mechanisms are currently integrated in the separate modules – every module produces its part of the execution log and it is up to the user to correlate the separate parts. If there is a discrepancy, the comparison module adds the details to the log and notifies the simulated SUT of the occurrence, which in turn provides further details as to what led to the event in the form of event history for the current execution and a model representation to map the event history to the current path.

However, it will make sense to extract and unify the logging mechanisms within a single dedicated logging module, which can then centrally manage all the necessary information and, if needed, request further information from the other modules. One could, for example, record the behavior of each test component separately, record different aspects of the behavior (functional, configurational, internal, external) separately, or cross-link these (e.g. by using vectors of "logical timestamps" [8]).

Another point to consider is the use of the test logging interface TCI-TL [7] provided by TTCN-3 – we could exploit it for our purposes, as it already provides comprehensive test execution logging. Currently it is not considered, but it could perhaps be tuned to serve some of our purposes as well.

4.6 Approach Summary

What we essentially presented is: First, insert specific statements in the test specification through instrumentation, to be able to get necessary information from within the test case at runtime. Then, during execution, the instrumented test case produces a simulation configuration that should exhibit a behavior close (or equal) to the behavior of the target SUT in the context of this test case. We take two such simulated SUTs and execute two test cases simultaneously, comparing them also simultaneously, on the fly, between defined *comparison points*. The test cases are executed repeatedly, until all possible behaviors that can be externally induced and observed have been covered. Should any discrepancies occur, a log will be maintained to facilitate quick and easy pinpointing of the causes for the differences in the test case behavior. Figure 4.22 sums up the overall process which reflects the conceptual architecture (Figure 4.23).

The whole approach was designed to be as close to fully automatic as possible. Because applying our approach can take a considerable amount of time, it will be rather undesirable to require manual user intervention every once in a while. Thus, the simulation, comparison

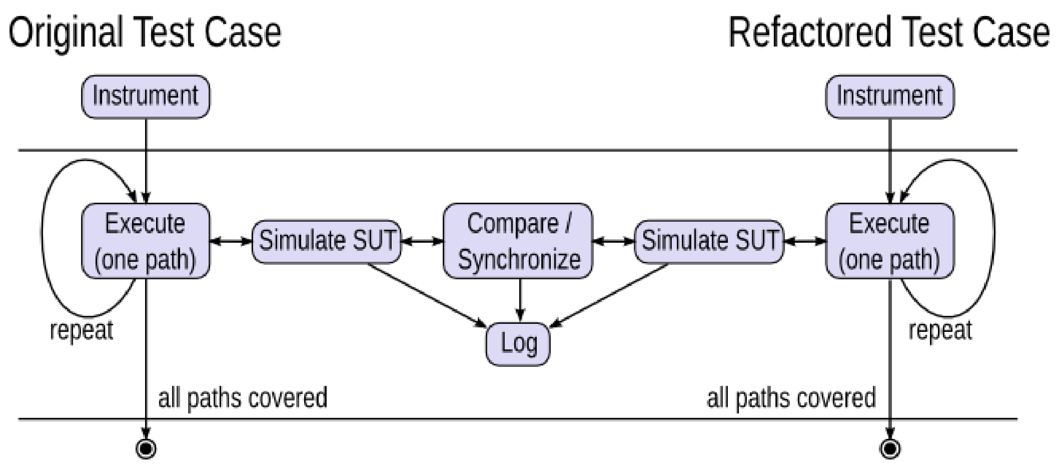

Figure 4.22: Equivalence checking process workflow summary

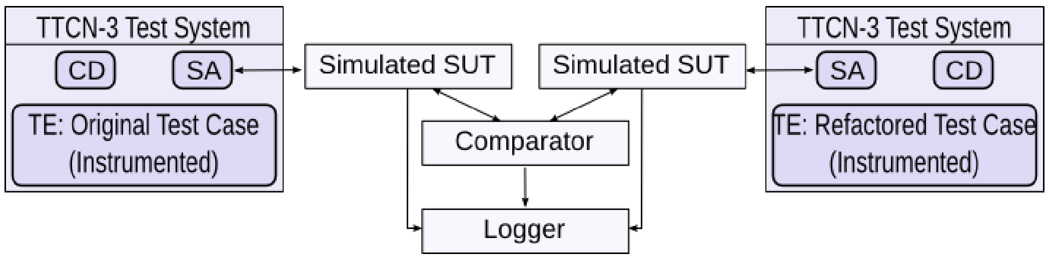

Figure 4.23: Conceptual architecture of the equivalence checking approach

and logging modules have been designed as reactive systems – interpreting received stimuli, sending back necessary stimuli, comparing traces and maintaining logs fully automatically. Only the preprocessing instrumentation step, which shall take the least time, may require some user interaction for input data in non-trivial situations.

5 Implementation

In this chapter we will outline some of the important aspects of our prototypical implementation of the approach described in Chapter 4. In general, we have four main conceptual functionalities that need to be implemented – instrumentation, simulation, comparison, and logging, which will be implemented in the instrumentor, simulator, comparator, and logger entities respectively. As already mentioned, the whole instrumentation topic, including its implementation, is covered in a separate technical report [12], and therefore we will focus only on the implementation of the remaining entities in the following sections of this chapter – simulator (Section 5.3), comparator (Section 5.4) and logger (Section 5.5). These entities are implemented as separate tools. Section 5.6 will present how they all fit and work together. But first, Section 5.1 will introduce a few requirements towards the test systems and Section 5.2 will introduce the Universal Message implementation specific concept.

5.1 Prerequisites

We have a few prerequisites the test system should be able to handle. First, test components should be uniquely identifiable, otherwise parallel simulation will not be possible. We have chosen to handle this particular issue using the standardized test component instantiation with a unique name parameter [4]. To ensure that this is indeed the case, we will generate and add unique name parameters to all the component instantiations during the instrumentation step[1,2]. Second, the simulator needs access to the codec implementation of the test system to allow direct use and manipulation of raw TTCN-3 templates and values. Most importantly, we need access to the SA implementation. Our simulator implementation has close ties to the SA and relies on it for essential information such as port resolution and addressing.

5.2 Universal Messages

To be able to fully understand our approach and its implementation, one needs to look under the hood of TTCN-3 test systems, and trace the message exchanging process in its entirety.

[1]The MTC is automatically instantiated with its name identifier "*mtc*".

[2]In some contexts it may be difficult to automatically assign unique names to test component instances – e.g. instantiation of multiple components in an array with loop

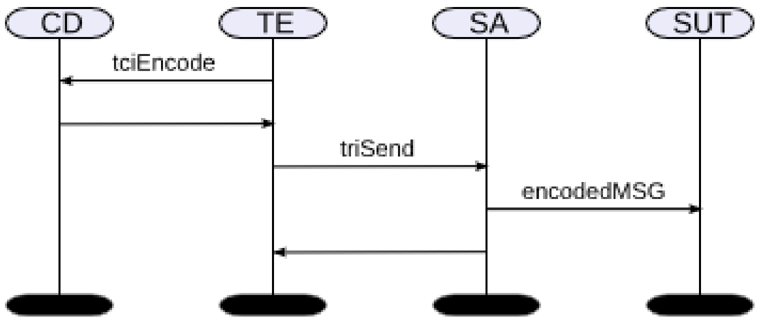

Figure 5.1: MSC illustrating the data flow during a send operation in TTCN-3

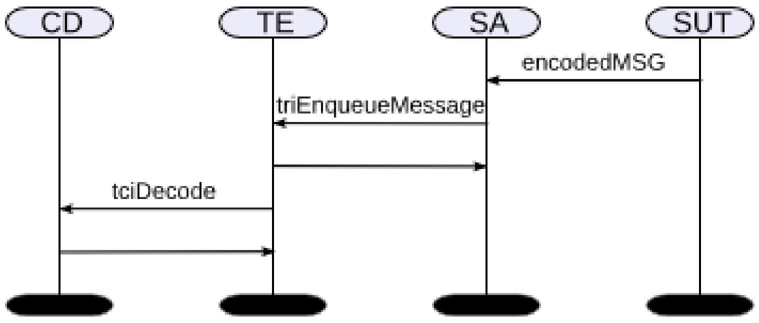

Figure 5.2: MSC illustrating the data flow during a receive operation in TTCN-3

We know so far that the actual behavior of a test case is executed within the core of a TTCN-3 test system - the test executable. But SUTs do not "speak" TTCN-3. Furthermore, the TTCN-3 test specification does not include any specifics concerning the communication medium. These two issues are handled by the codec and the SA in a TTCN-3 test system respectively. Thus messages that have to be sent to the SUT are first encoded within the CD, then passed on to the SA, which manages the addressing and actual communications with the SUT and finally arrive at the correct address[3] at the SUT. The same way is taken in the reverse direction when messages are being sent from the SUT - messages received at the SA are forwarded to the target test components and pass through the CD where they are decoded back to TTCN-3 data and passed on to the TE. The data flow towards the SUT and back is illustrated on Figures 5.1 and 5.2 respectively.

This could be analogously illustrated with another example – picture an English executive that needs to send a message to China and neither the executive, nor his Chinese correspondents speak each other's languages. So the executive passes his message to the

[3]"Address" and "addressing" are used here as general terms, not to be confused with SUT-addresses in TTCN-3. Thus, in this context "addressing" includes both system ports and system addresses.

translator's office, who translates is into Chinese and then passes it on to the secretary or messenger's office, who then arrange the actual sending of the message, with the correct recipient and return address, over the specific medium - be it post, e-mail, dedicated courier and await replies over it. Then upon receipt of the reply is addressed back to the proper executive (there may be more than one, sharing the same secretary's office), but the executive still cannot make anything out of it since the reply is in Chinese, therefore the message is redirected to the translator's office and finally returns to the executive in his native English.

We will build upon this interaction mechanism. First, we will introduce a universal data carrier, named accordingly, a Universal Message (UM). There are already many data types already. However, the data types used within the TTCN-3 test system have specific purposes and are limited to a particular scope. The native TTCN-3 data types are used within the TTCN-3 test case specifications and can be passed on to the CD for encoding and decoding. Encoding TTCN-3 data yields what is referred to in the TTCN-3 standards as "*triMessages*" (or simply the encoded TTCN-3 data). Decoding is the opposite process and produces TTCN-3 "*values*" (TTCN-3 data) out of *triMessages*. The SA on the other hand operates on *triMessages* only and has no knowledge of the underlying *values*. While this is perfectly logical in the normal usage context, where the SUT need not know anything about the underlying TTCN-3 data, for our purposes it may become necessary, that the *values* can be correlated to the *triMessages* and vice-versa. This is one of the purposes of the UM that we just introduced. It serves as a package to contain the TTCN-3 *value*, its encoded version and further information as necessary.

During encoding one such message is instantiated and both the original *value* and the encoded version are put in it. Then, it is passed on to the SA. The newly introduced UMs will be used for unified data exchange between the test system and the simulated SUT, and within them as well. The normal data exchange mechanisms within the test system will be left untouched, to prevent interference with its operation. However, they will be exploited indirectly for our purposes.

One of the specific reasons to maintain such correlation is to simplify, or even completely circumvent decoding. Under normal circumstances, implementing decoders for TTCN-3 can be a rather complicated task, depending on the complexity of the data types and their diversity. As we stated in our approach description, for the purposes of simulation, the expected messages are sent from inside the test specification (with the help of instrumentation), and then sent back from the simulated SUT with the appropriate target port of the test system, at the appropriate moment. Now, in such a context, it makes a lot of sense to keep the original TTCN-3 *value* and correlate it to a key – the encoded *triMessage* in this context. When (if)[4] this same message comes back later, its original TTCN-3 *value* can simply be looked up using its key (which is what the simulated SUT sends back), and the

[4]In the context of alt statements, only one of multiple possible messages is sent back.

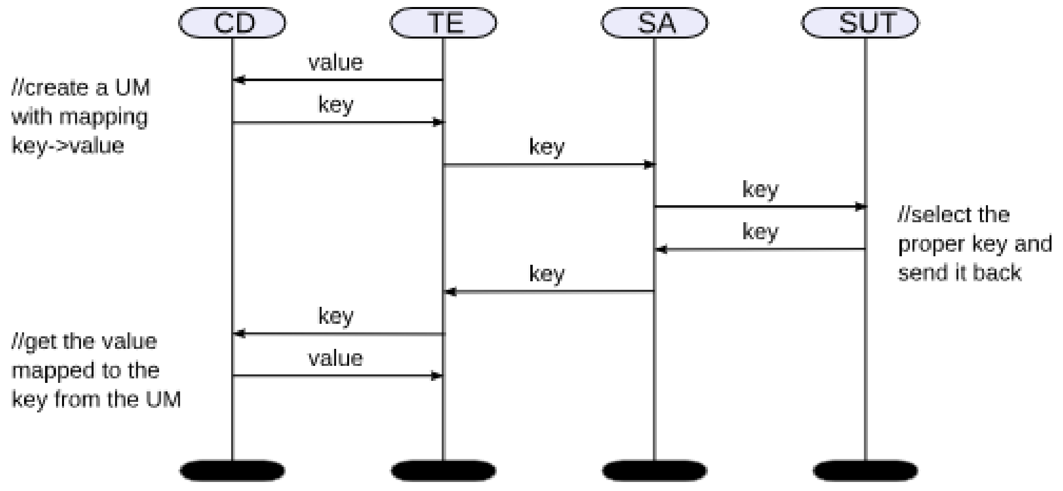
CD
TE
SA
SUT
value
//create a UM with mapping key->value
key
key
key
key
//select the proper key and send it back
key
key
//get the value mapped to the key from the UM
value

Figure 5.3: MSC illustrating the usage of UMs to circumvent decoding in our approach

readily available TTCN-3 *value* can then be passed back to the TE. Figure 5.3 illustrates the process.

Next, in the SA we enrich the UM, adding transport data, such as source test component, target port and address and pass it on to the simulated SUT, whose implementation we will be referring to as simply the *simulator* from now on, to distinguish it from the concept. Note that UMs are not exchanged parallel to the standard messages between the TTCN-3 functional entities, and with the simulator. Conceptually, it is the case, but technically, the UMs are kept in a pool within the simulator and only the keys to these messages are passed on, which conveniently enough are the encoded messages produced by the encoding process in the beginning. Such packages are also used for convenience, making the data structures and associated contents universally accessible by a single reference key. They can also be used to re-associate and repackage existing data, such as a adding target ports when expected messages are sent back.

For all these purposes, we exploit the native TTCN-3 operations provided by the SA and the codec with minor modifications to accommodate our specific needs. Furthermore, we also make use of the mapping and unmapping operations of the SA, to monitor the current port mappings during test case execution, and subsequently to successfully resolve the correct communication channels.

So far we have described how we can get operational data from within the test system. What happens next is the interpretation of this data, to drive our simulator.

5.3 Simulator

The simulator is implemented as a two-part entity – comprising of a client and a server. This is mainly for flexibility, but also for task separation. The client manages the runtime data which includes interpretation of control messages, port mappings and their resolution, test component instances and their state, implementation of the synchronization mechanisms as described in Section 4.3.7, although they could also be implemented within the server. It maintains a behavior representation of the current test case execution, using an implementation of the LTS behavior models described in Section 4.1.1. The client has complete knowledge of the current execution, but none of the previous or subsequent executions, because it is reinitialized for every subsequent test case execution. Therefore, it does not have a decision mechanism to react automatically. It implements the interactive simulation concept, providing an interface to the user for manual selection of stimuli, and to the server for an automated one. Its communication with the server is realized through the exchange of *action keys*, which serve as the labels on the LTS-based behavior models. The *action keys* are associated with specific actions that are taken or can be taken, similar to the functional traces described in Section 4.3.5. This way, it helps the server build its own persistent representation of the behavior model, which is used for exhaustive coverage of the test case behavior during subsequent executions. The simulation client also produces the traces for comparison and analysis, and interacts with the comparator.

The simulation server is the "brain" of the simulator. It maintains persistent behavior representations during and between executions, which are constantly changing, being expanded with subsequent executions as further data becomes available and reduced when parts of them are no longer required. The data, as mentioned, is reduced to *action keys*. They are enough to model the interactive behavior and trigger specific actions in the simulation client, such as sending a specific message over a specific port.

The major task of the server is to implement the path-finding algorithm described in Section 4.3.6, to systematically provide the "*first available action*" of the simulated SUT for a specific test component on a client request. This is facilitated by the backtracking algorithm that tags the already explored paths at the end of an execution on client notification. The server also notifies the client when all paths of the test case behavior have been covered. An individual server instance is used to operate every[5] simulation client, which ensures that the simultaneously executing test cases are simulated independently from one another. Alternatively, the servers can be configured to take over each other's control.

The whole simulator itself is designed as a standalone tool, to provide interactive or automated simulation for generic purposes, outside the specific context of this thesis.

[5]Current design uses a two-server configuration and thus supports only two simultaneous automated simulations on its two interfaces. This has been in accordance with our specific task. However, with a little redesign, more simultaneous automated simulations can be made possible, if a need there may be

The behavior representations are a rather flexible implementation of an LTS, with some extensions for our purposes. They are implemented as sets of nodes and edges with relations binding them together. The edges are associated with *action keys*, or labels in the general sense. Additionally, nodes and edges have boolean "*checked*" flags to facilitate the backtracking and pathfinding algorithms as described in Section 4.3.6.

The individual backtracking algorithm operates on behavior representation models and is implemented within the model implementation. The global multi-component backtracking algorithm is controlled externally, by the simulation server. Pathfinding is realized on the fly during test case execution, through the simulation server.

5.4 Comparator

The comparison module is also implemented as a standalone tool, comprising of two parts as well - a comparison client and a comparison server. The comparison client provides a simple interface to the simulation clients for bidirectional communication with the comparison server. The communication must be bidirectional, since the server shall be able to notify the simulator on key events, such as comparison results or the release of component locks. The comparison client will then pass such events on to the simulator client. There is one comparison client instance for each simulation client instance.

The comparison server on the other hand performs the bulk of the comparison job. It has two comparison interfaces that reflect the TSI structures (individually for both test systems) and their respective hierarchies, with all the corresponding port instances and their respective incoming and outgoing queues. It classifies the messages hierarchically and compares the corresponding structures of the two system interfaces. Messages are compared in their encoded representations. Control messages are not compared with the exception of substitutes for discrete events, such as timer substitutes. Additionally, it provides a direct interface to its comparison functionalities, for the comparison of data other than comparison traces.

5.5 Logger

The logging mechanisms are currently implemented within the separate modules and produce logging information as required during their execution, which is then collected in five main pools – one for each simulation client and each simulation server, and one for the comparison server. In future developments, all modules will have unified logging interfaces to interact with a dedicated logging module which will centrally manage all the logging information, realized also using a client-server model.

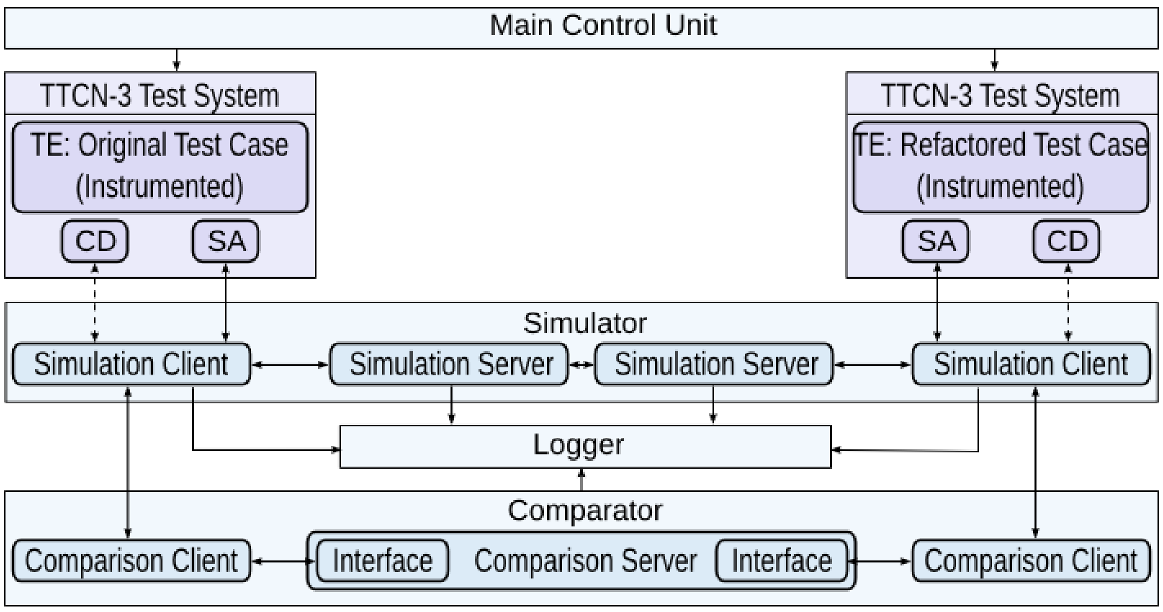

Figure 5.4: Implementation architecture of our approach

5.6 Overall

Additionally, we have designed a loading module or a "*main control unit*", which puts all the pieces together. It has the task of continuously re-executing the test case through the provided management interface from the TTCN-3 tools until the simulator notifies it that all behaviors have been covered. It also manages the comparison and simulation modules. Currently it is just a tool to automate the tasks during development and testing, but it has the prospects to become the main user interface towards our complete system and all the individual tools.

To complete the picture, we will use a few diagrams to illustrate how everything fits together. Figure 5.4 illustrates the overall implementation architecture and Figure 5.5 illustrates where the individual parts of our approach and their co-relations fit in the context of the standard TTCN-3 architecture.

All the modules for the prototypical implementation of our approach are implemented in JAVA. The client-server models are realized using the JAVA Remote Method Invocation (RMI) for simplicity.

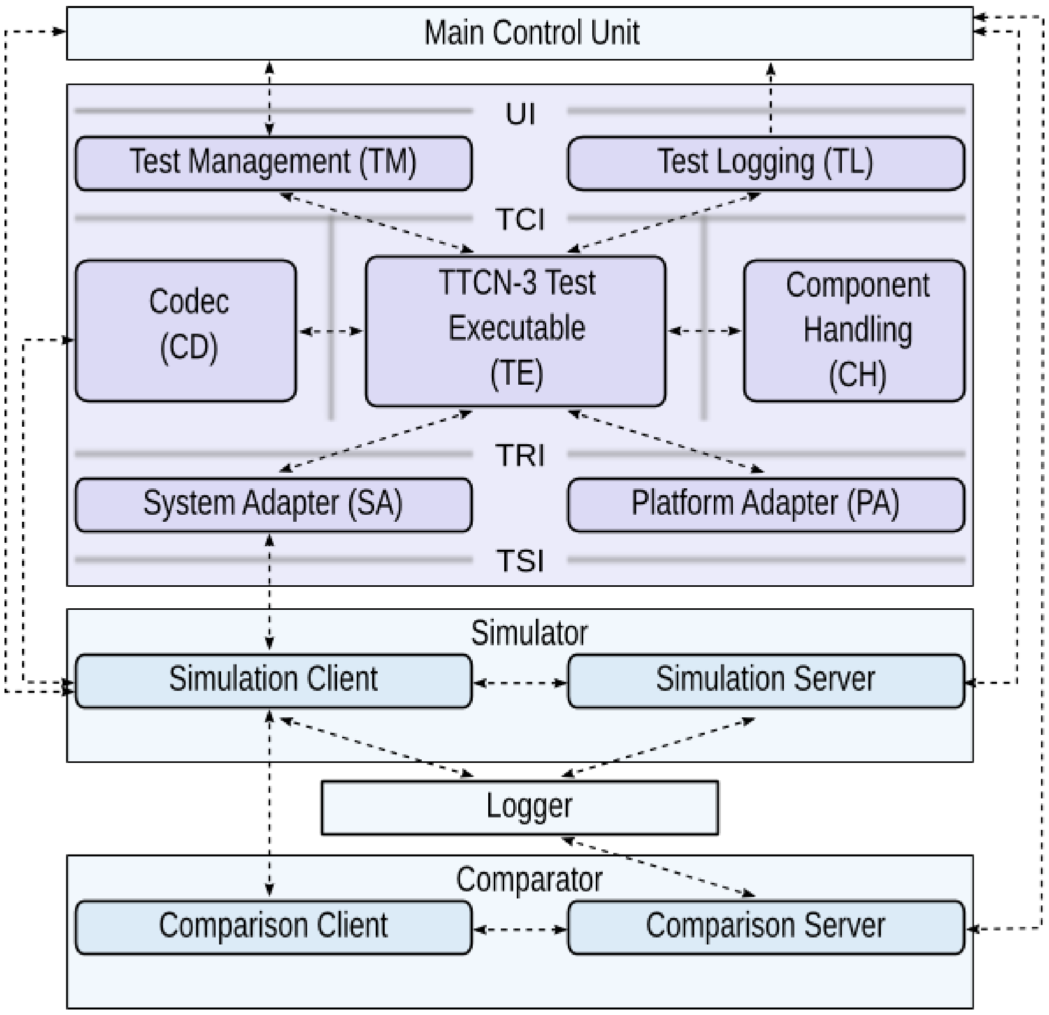

Figure 5.5: Our approach in the context of the TTCN-3 standard architecture

6 Case Studies

In this chapter, we will present six case studies illustrate and test the applicability of our approach and its prototypical implementation. But first, we will elaborate more on the types of changes to behavior as outlined in Chapter 3.

We observe and analyze the test case behavior through data. Therefore, data flow and data integrity play a significant role in the observable test case behavior. If the wrong data is exchanged, the test case behavior is invalidated. Apart from that, data is usually used to control the execution of a test case and thus often affects its behavior directly.

Three basic types of errors were outlined in Chapter 3. They can be correlated to the types of anomalies that introduced them as follows:

- Wrong message contents error generally corresponds to corrupted data descriptions and/or data flow.
- Wrong message order error in turn suggests corrupted control flow
- Wrong communication points error can imply corrupted configuration, data description changes (if port and component references are perceived as data) or control flow changes (if a configuration change has been moved through refactoring) Additionally, if messages are skipped altogether (which falls under the wrong message ordering error), corrupted control flow is the most likely cause.

We will not state explicitly which type of error has occurred and which anomaly had caused it. Instead, we will focus on the implications of eventual errors in the given examples.

The nature and purpose of refactoring as a process severely limits our options for case studies. If a refactoring is determined successfully applied, the refactored version supersedes the original, leaving no traces to the user other than artifacts in the version control system. Both the original and refactored versions of a test case, or any other software for that matter, are usually only available to the developers performing the refactorings on them. Since there are no open-source test suites undergoing heavy development that can be considered, we have rather restricted options on how to proceed with our experiments. We could take selected test cases from publicly available test suites, which (probably) already feature all the necessary refactorings and try to apply further refactorings on them. We could also try to apply refactorings in the reverse direction, to see what an earlier version of the test case might have been like and use it as our base version for comparison.

However, the data will need additional explanation and can be more confusing than useful for the illustration of the application of our approach. We are interested in the particulars of correctly and incorrectly applied refactorings. To keep the case studies simple and understandable, we will analyze a few refactorings using custom sample test cases to illustrate the correct and incorrect application of these refactorings. The refactorings are taken from Zeiss [29]. The case studies will be structured as follows:

Synopsis – A short summarized information about refactoring along with a reference to the detailed descriptions and application steps available in [29].

Possible Issues – Possible problems that may occur due to the incorrect application of the given refactoring and further details one should be careful about when applying the refactoring.

Application Examples – Analysis of correct and incorrect application examples for the given refactoring, accompanied by results of the application of our approach to determine whether the observable behavior has been preserved. We are interested in two things – whether our approach can be used to prove if a refactoring was applied successfully, and if it was not, whether our approach can be used to prove that as well.

In the application examples, we will refer to the test case prior to refactoring as “original” and after refactoring as “refactored”. The rest of this chapter includes six case studies using the following refactorings from [29]:

- Extract Altstep
- Replace Altstep with Default
- Replace Template with Modified Template
- Parametrize Template
- Prefix Imported Declarations
- Extract Function

6.1 Extract Altstep

Synopsis: Move identical alternative branches of alt statements to an altstep when used more than once ([29], Section 4.2.1, p.35).

Possible Issues: Parameter passing may become an issue, especially with timers[1]. Parameter passing is a very sensitive issue, as apart from passing the correct parameters, they also need to be passed on with the correct modes (as *in*, *out* or *inout* parameters) and failing to do so may induce some issues. Not taking branch guards into consideration will also most certainly be an issue.

Application Examples: Given the simplified sample test case in Listing 6.1, we can identify repeated alternative branches than can be extracted as an altstep to improve reusability and maintainability. Listing 6.2 subsequently shows the refactored version. We have extracted the *mtcPort1.receive(msgC)* branch to an altstep called *alt_Fail*.

```
testcase tc1() runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  mtcPort1.send(msgA);

  alt{
    [] mtcPort1.receive(msgB) {
      setverdict(pass);
      mtcPort1.send(msgD);
    }
    [] mtcPort1.receive(msgC) { //repeated branch
      setverdict(fail);
    }
  }

  mtcPort1.send(msgE);

  alt{
    [] mtcPort1.receive(msgF) {
      setverdict(pass);
      mtcPort1.send(msgA);
    }
    [] mtcPort1.receive(msgC) { //repeated branch
      setverdict(fail);
    }
  }

  mtcPort1.send(msgE);

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.1: Sample test case pending an Extract Altstep refactoring

[1]Zeiss [29] mentions that ports have to be passed as parameters (*inout* parameters to be precise). Ports actually do not have to be passed as parameters at all to altsteps, since they are never local. Ports are always defined within the test component definition scope. But if we extract an altstep in a generic manner, without a *runs on* clause, and we extracted a common alternative branch from the behaviors of components from different types, the altstep will need to have the ports of the specific test components passed as parameters.

```
altstep alt_Fail () runs on mtcType{
  [] mtcPort1.receive(msgC) {
    setverdict(fail);
  }
}

//...

//tc1 refactored using the
//extract altstep refactoring
testcase tc1_refactored() runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  mtcPort1.send(msgA);

  alt{
    [] mtcPort1.receive(msgB) {
      setverdict(pass);
      mtcPort1.send(msgD);
    }
    [] alt_Fail () {}
  }

  mtcPort1.send(msgE);

  alt{
    [] mtcPort1.receive(msgF) {
      setverdict(pass);
      mtcPort1.send(msgA);
    }
    [] alt_Fail () {}
  }

  mtcPort1.send(msgE);

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.2: The sample test case in Listing 6.1 refactored using an Extract Altstep refactoring

Applying our approach to validate both versions produces the following results: First, we build the behavior models (from the SUT's perspective) for the single test component MTC to be able to simulate the execution of the test case and follow the steps taken during equivalence checking. In Figure 6.1 (a) and (b), we can see the behavior model representation and the traces for comparison generated at the comparison points during the first execution of the original and the refactored test cases respectively. Plotting the traces against each other, we can easily see that they are identical and after comparing the verdicts at the end of the execution (**pass**, **pass**), we can consider the behavior during the first execution equivalent. Figure 6.2 illustrates the situation during the second execution. Again, nothing peculiar has occurred – the traces produced are the same and so are the verdicts. Figures 6.3 and 6.4 show the traces for comparison for the remaining two possible executions, and after comparing

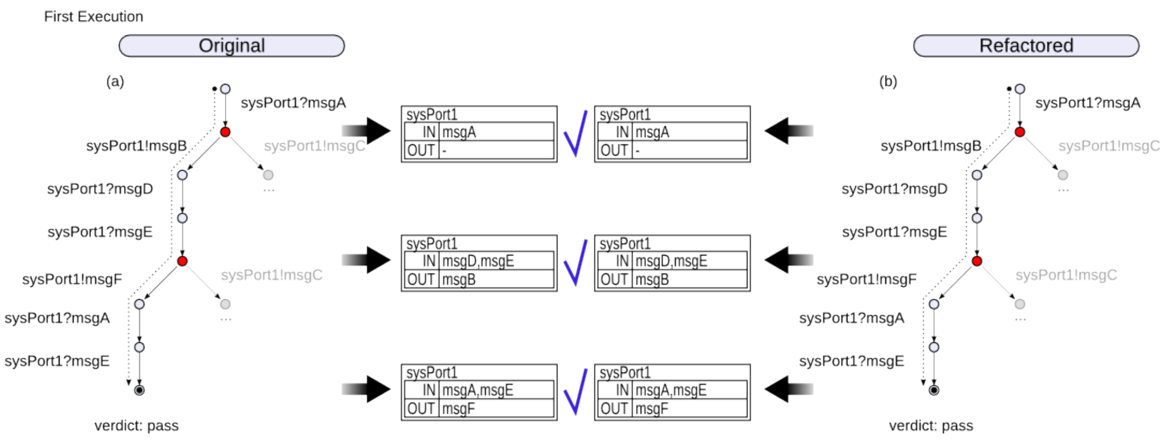

Figure 6.1: Behavior representation and traces for comparison during the first execution of (a) the original test case as specified in Listing 6.1, and (b) the refactored test case as specified in Listing 6.2.

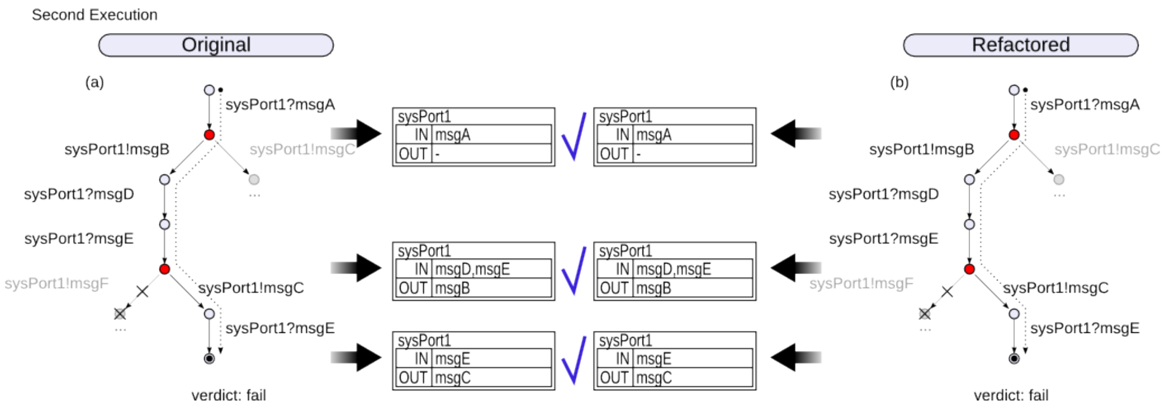

Figure 6.2: Behavior representation and traces for comparison during the second execution of (a) the original test case as specified in Listing 6.1, and (b) the refactored test case as specified in Listing 6.2.

the verdicts produced at the end of each, we can conclude that the observable behavior is preserved in all possible executions of this test case and thus the application of the extract altstep refactoring in this case is successful.

```
1  testcase tc1v2() runs on mtcType system systemType{
2    map(mtc:mtcPort1, system:systemPort1);
3
4    mtcPort1.send(msgA);
5
6    alt{
7      [] mtcPort1.receive(msgB) {
8        setverdict(pass);
9        mtcPort1.send(msgD);
10     }
11     [] mtcPort1.receive(msgC) {
12       mtcPort1.send(msgD); //additional action
13       setverdict(fail);
14     }
15   }
16
17   mtcPort1.send(msgE);
18
19   alt{
20     [] mtcPort1.receive(msgF) {
21       setverdict(pass);
22       mtcPort1.send(msgA);
23     }
24     [] mtcPort1.receive(msgC) {
25       setverdict(fail);
26     }
27   }
28
29   mtcPort1.send(msgE);
30
31   unmap(mtc:mtcPort1, system:systemPort1);
32 }
```

Listing 6.3: Alternative version of the sample test case in Listing 6.1 with an additional action in the scope of the extracted branch

Now, suppose there was an additional **send** statement (Listing 6.3) in the scope of the extracted branch in the first instance (Line 12), but not the second. It would be rather obvious in this simplified example, but may go unnoticed in multiple occurrences of similar branches with small differences. Depending on which instance of the supposedly identical branch we take for the extracted altstep the error will occur during the second and the fourth or during the third execution of our sample test case. If we take the first instance (Listing 6.4), then our altstep will contain this additional send statement and it will occur when we select the altstep branch during the second and the fourth executions (the refactored test case *tc1v2_refactored* will be identical to *tc1_refactored* in Listing 6.2, with only the name of the extracted altstep being different – *alt_Fail* substituted through *alt_Failv2*). We will only illustrate the second

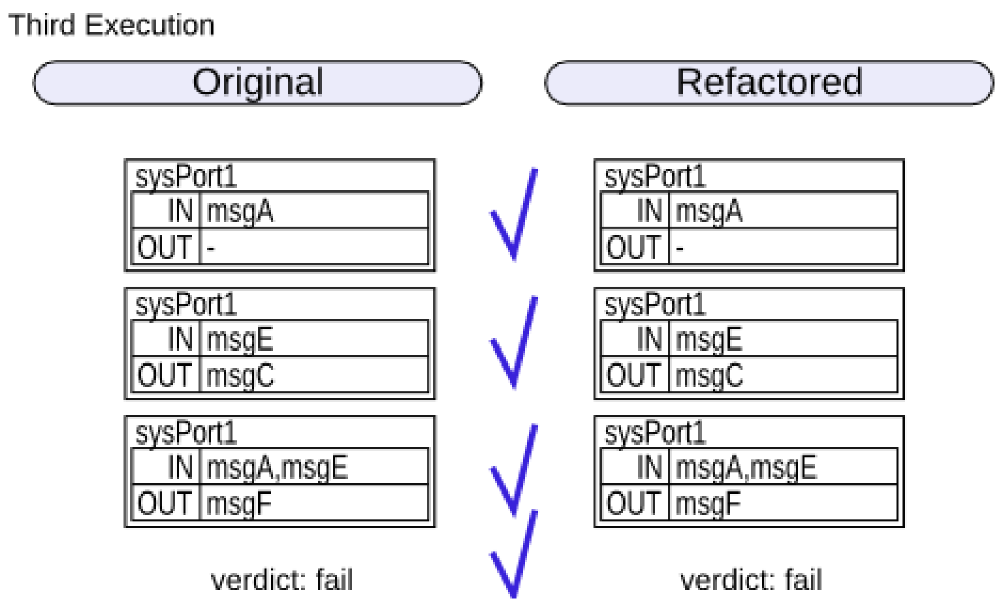

Figure 6.3: Comparison of the traces generated during the third execution of the original and refactored test cases in Listings 6.1 and 6.2 respectively.

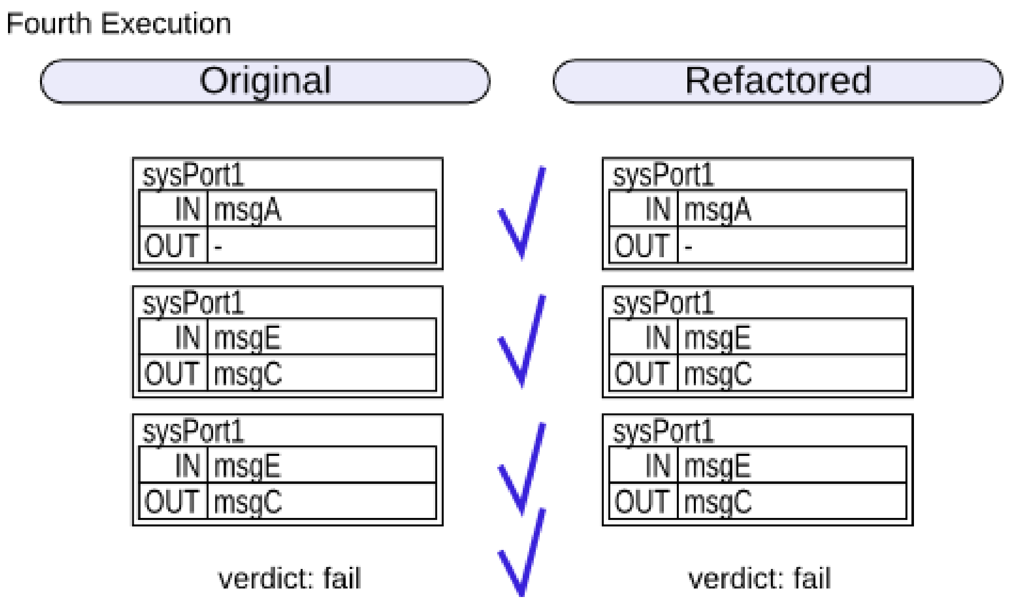

Figure 6.4: Comparison of the traces generated during the fourth execution of the original and refactored test cases in Listings 6.1 and 6.2 respectively.

and the fourth execution (Figures 6.5 and 6.6 respectively), since they are the only relevant ones, where the differences in the observable behavior become apparent. The errors are highlighted in red.

```
altstep alt_Failv2 () runs on mtcType{
  [] mtcPort1.receive(msgC) {
    mtcPort1.send(msgD); //extracted the branch with the additional action
    setverdict(fail);
  }
}
```

Listing 6.4: Incorrectly extracted altstep from alternative version of the sample test case in Listing 6.3 with an additional action in the scope of the extracted branch

```
testcase tc1v3() runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  var integer failures := 0;
  mtcPort1.send(msgA);

  alt{
    [] mtcPort1.receive(msgB) {
      setverdict(pass);
      mtcPort1.send(msgD);
    }
    [] mtcPort1.receive(msgC) {
      setverdict(inconc);
      failures := failures +1;
    }
  }

  mtcPort1.send(msgE);

  alt{
    [] mtcPort1.receive(msgF) {
      setverdict(pass);
      mtcPort1.send(msgA);
    }
    [] mtcPort1.receive(msgC) {
      setverdict(inconc);
      failures := failures +1;
    }
  }

  mtcPort1.send(msgE);

  if (failures > 1) {
    setverdict(fail);
  }

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.5: Alternative version of the sample test case in Listing 6.1 with a local variable to count the failures

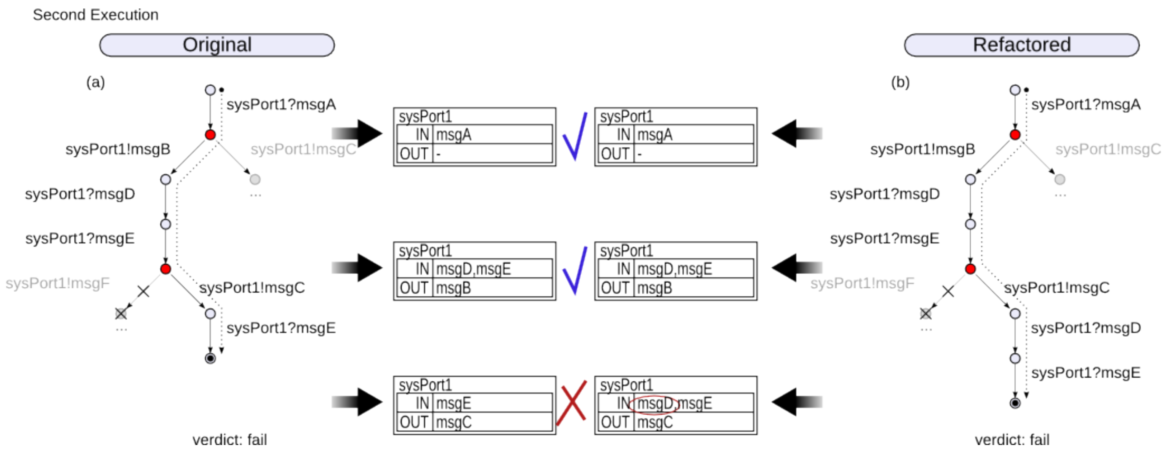

Figure 6.5: Behavior representation and traces for comparison during the second execution of (a) the modified original test case as specified in Listing 6.3, and (b) the incorrectly refactored test case using the incorrectly extracted altstep in Listing 6.4.

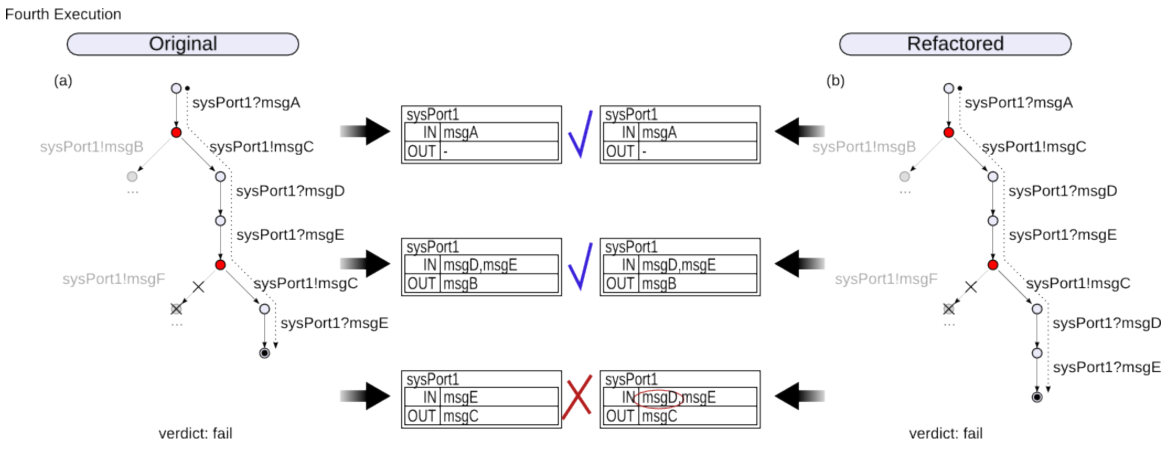

Figure 6.6: Behavior representation and traces for comparison during the fourth execution of (a) the modified original test case as specified in Listing 6.3, and (b) the incorrectly refactored test case using the incorrectly extracted altstep in Listing 6.4.

Additionally, failing to take into consideration local variables that may have changed during execution from the one occurrence of the identical alt branch to another and have an impact on the behavior or the data being exchanged can lead to even more striking discrepancies. Listing 6.5 illustrates such a case. It is a modification of our original sample test case (Listing 6.1), where the verdict is set to **fail** only if both interactions failed (message *msgC* was received), with the help of a variable to count the failures. If we do not consider the variable during altstep extraction (Listing 6.6), a compilation error will occur. If we pass it by value (Listing 6.7), the test case will terminate with verdict **inconc** instead of **fail**, even though it interacts with the SUT in the same way, because the changes to the variable *failures* will not be visible outside the scope of the altstep and when it is evaluated at the end (Line 33 in Listing 6.5), its value will still be the initial value of '0'. Similar situations may also lead to different interactions. If the variable was passed with the **out** mode only it should also result in a problem – its value should be reset every time the altstep is applied. Experiments with TTCN-3 tools have shown, however, that it will be treated as if it were passed with the **inout** mode. The TTCN-3 standard [4] refers to the value of the parameter as unitialized or "unbound" in such cases, which suggests that the value of *failures* should be reset in this case, before being incremented, thus resulting in '1' as its final value.

```
altstep alt_Failv3_1 () runs on mtcType{
  [] mtcPort1.receive(msgC) {
    setverdict(inconc);
    failures := failures +1;
  }
}
```

Listing 6.6: Incorrectly extracted altstep without parameter passing of local variables from Listing 6.5.

```
altstep alt_Failv3_2 (in integer failures) runs on mtcType{
  [] mtcPort1.receive(msgC) {
    setverdict(inconc);
    failures := failures +1;
  }
}
```

Listing 6.7: Incorrectly extracted altstep with parameter passing by value of local variables from Listing 6.5.

Our approach will most likely be able to identify all errors introduced through the incorrect application of the *Extract Altstep* refactoring, should they occur. There are incorrect applications of this and other refactorings that will not manifest themselves in the form of directly observable changes to behavior. Changes to behavior that are not directly observable cannot be automatically detected using our approach.

6.2 Replace Altstep with Default

Synopsis: Identical altstep branches at the end of subsequent alt statements can be replaced by a default altstep activation ([29], Section 4.2.3, p.41). The *Replace Altstep with Default* refactoring is the logical next step after applying the *Extract Altstep* refactoring where the context permits. However, only altsteps containing parameters passed by value can be activated. This especially affects the usage of timers, as they must always be passed by reference (using the **inout** mode). Thus, local timers cannot be passed as arguments to activated default altsteps. Fortunately, timers declared within the test component definition can be used instead (provided that the altstep is component type specific - defined with a runs on clause).

Possible Issues: Given that the altsteps are extracted correctly, there are a number of specific issues that may occur due to incorrect application of the *Replace Altstep with Default* refactoring. Since activated defaults attach to **all** blocking statements, they should be used with caution, as deeply nested blocking statements, where the default altstep should not be considered, will inherit the branches of the activated default and thus provide invalid alternatives at such points. Replacing an altstep which is not the last branch of an alt statement will also produce a change in the observable behavior. The order of activation of multiple defaults should also be very carefully addressed.

Application Examples: In the following, we will illustrate briefly (Listing 6.8) the application of the *Replace Altstep with Default* refactoring to the correctly refactored version of our example. Then, we will compare it to the original version (Listing 6.1) and the later refactored version from which it was derived (Listing 6.2).

Applying our approach to this version of the test case will produce the model illustrated in Figure 6.7, which would essentially result in an identical behavior representation model and comparison traces (Figure 6.8) as the ones resulting from the original (Figure 6.1 (a)) and the refactored (Figure 6.1 (b)) test cases from which this refactored version was derived[2]. So when comparing the traces produced from this version to the traces from the other refactored version (Listing 6.2), during the second (Figure 6.9), third (Figure 6.10) and fourth (Figure 6.11) executions, which are the only relevant ones in this case (although applying the implementation of our approach will still need to compare all four possible executions), there will be no differences detected and both versions will be considered equivalent.

[2]Only the behavior representation models during the first executions are illustrated on the given figures. The complete behavior representation models are also identical.

```
testcase tc1_refactored_v2() runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  mtcPort1.send(msgA);

  var default alt_Fail_reference := activate(alt_Fail());

  alt{
    [] mtcPort1.receive(msgB) {
      setverdict(pass);
      mtcPort1.send(msgD);
    }
  }

  mtcPort1.send(msgE);

  alt{
    [] mtcPort1.receive(msgF) {
      setverdict(pass);
      mtcPort1.send(msgA);
    }
  }

  deactivate(alt_Fail_reference);

  mtcPort1.send(msgE);

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.8: A further refactored version of the correctly refactored version (Listing 6.2) of our sample test case (Listing 6.1) using the Replace Altstep with Default *refactoring.*

We can also compare this version to the original version of the test case to illustrate the transitivity of the bisimulation relation and our approach in particular. This will not be necessary, as the sample case is rather trivial and it is obvious that the equivalence between all three versions holds under all circumstances.

Now consider a variation of our sample test case, which has already undergone the *Extract Altstep* refactoring, but has a nested standalone *receive* statement (Listing 6.9, Line 10).

```
1  testcase tc1v4_refactored () runs on mtcType system systemType{
2    map(mtc:mtcPort1, system:systemPort1);
3
4    mtcPort1.send(msgA);
5
6    alt{
7      [] mtcPort1.receive(msgB) {
8        setverdict(pass);
9        mtcPort1.send(msgD);
10       mtcPort1.receive(msgE); //additional blocking statement
11     }
12     [] alt_Fail () {}
13   }
```

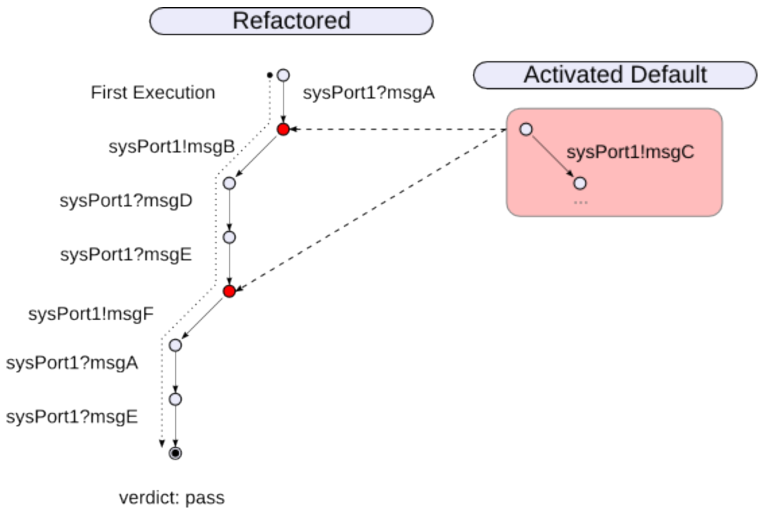

Figure 6.7: Intermediate behavior representation during the first execution of the correctly refactored test case using the Replace Altstep with Default *refactoring in Listing 6.8.*

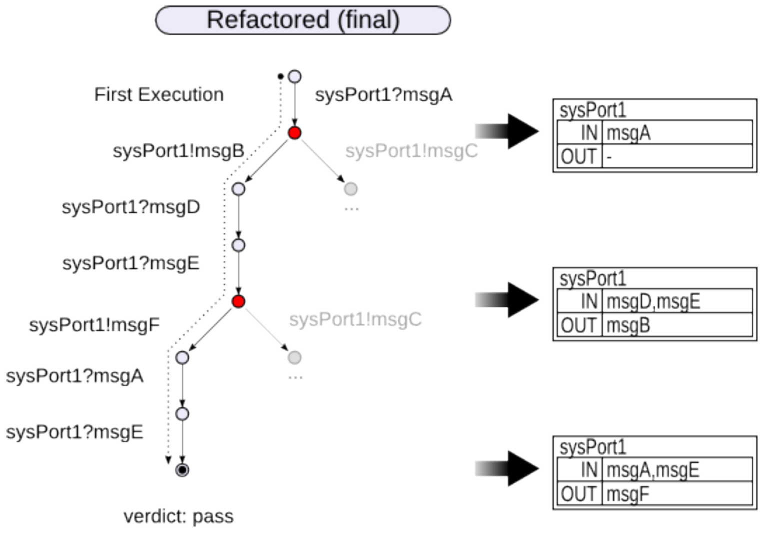

Figure 6.8: Final behavior representation and comparison traces during the first execution of the correctly refactored test case using the Replace Altstep with Default *refactoring in Listing 6.8 constructed from Figure 6.7.*

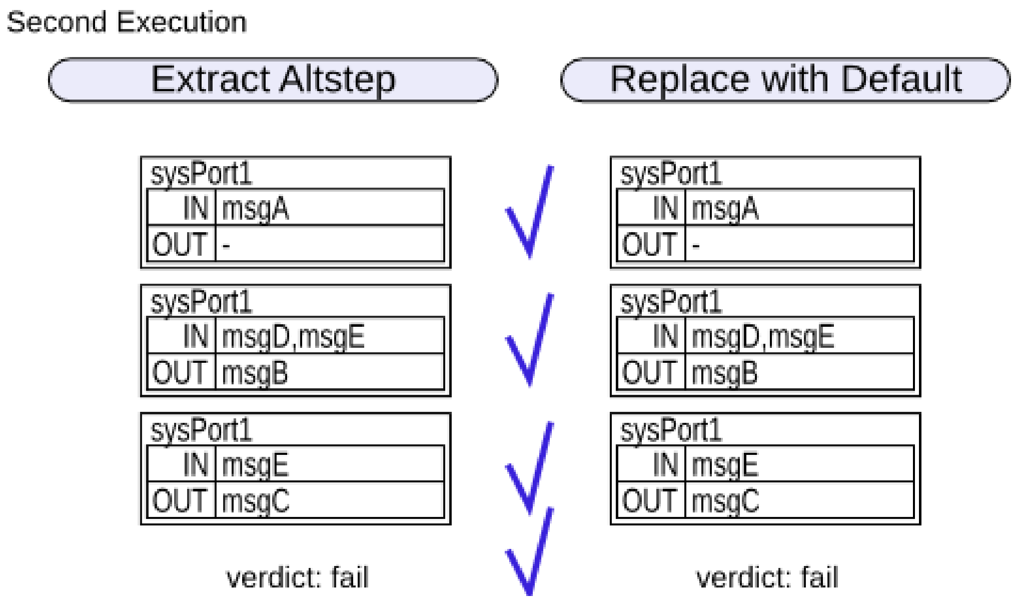

Figure 6.9: Comparison of the traces generated during the second execution of the test cases refactored using the Extract Altstep *refactoring (Listing 6.2) and subsequently* Replace Altstep with Default *refactoring (Listing 6.8) refactorings on the original test case in Listing 6.1.*

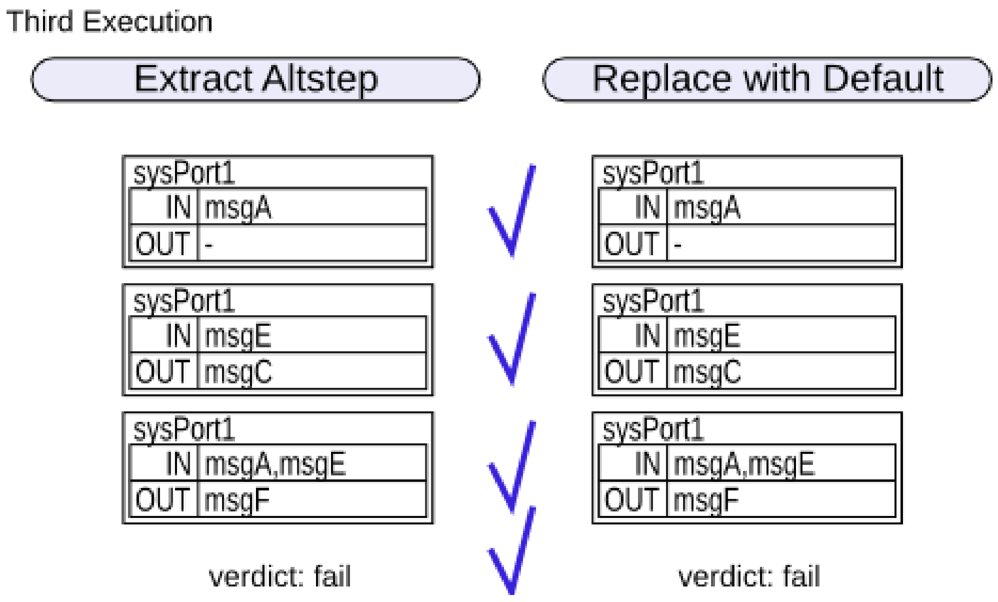

Figure 6.10: Comparison of the traces generated during the third execution of the test cases refactored using the Extract Altstep *refactoring (Listing 6.2) and subsequently* Replace Altstep with Default *(Listing 6.8) refactorings on the original test case in Listing 6.1.*

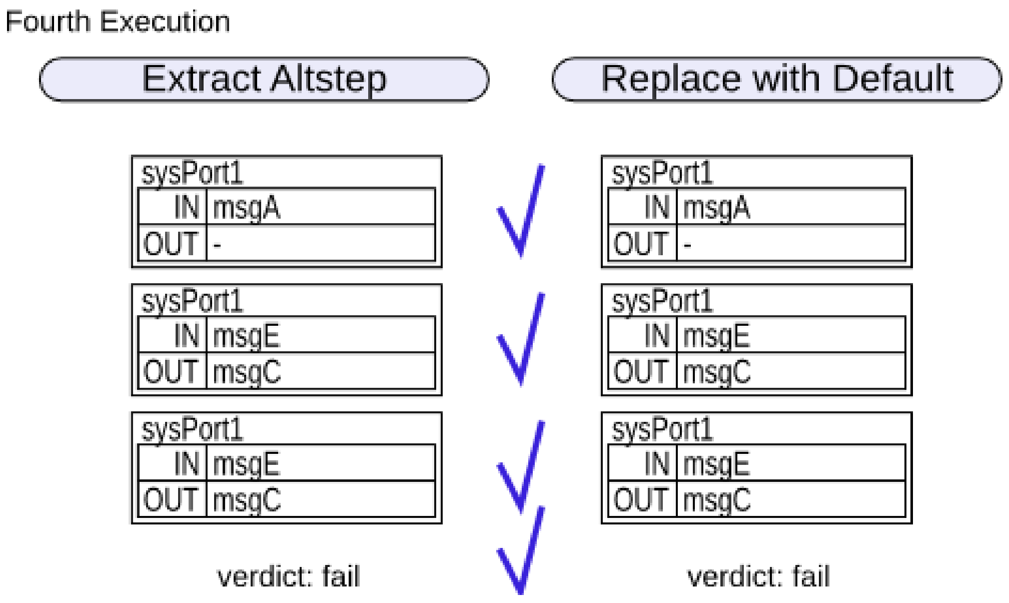

Figure 6.11: Comparison of the traces generated during the fourth execution of the test cases refactored using the Extract Altstep *refactoring (Listing 6.2) and subsequently* Replace Altstep with Default *(Listing 6.8) refactorings on the original test case in Listing 6.1.*

```

    mtcPort1.send(msgE);

    alt{
      [] mtcPort1.receive(msgF) {
        setverdict(pass);
        mtcPort1.send(msgA);
      }
      [] alt_Fail () {}
    }

    mtcPort1.send(msgE);

    unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.9: A modified version of Listing 6.2.

If we apply the *Replace Altstep with Default* refactoring, neglecting the fact that there is a nested blocking statement (Listing 6.10, Figure 6.12), when we apply our approach to check the original (Figure 6.13) and the newly refactored (Figure 6.14) test cases for equivalence with the help of our approach, we notice that there is a difference in the behavior representations. However, our approach will first notice the difference during the third execution (Figure 6.15), when the newly introduced branch is accessed, which in this case will also result in two additional executions.

```
testcase tc1v4_refactored_v2() runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  mtcPort1.send(msgA);

  var default alt_Fail_reference := activate(alt_Fail());

  alt{
    [] mtcPort1.receive(msgB) {
      setverdict(pass);
      mtcPort1.send(msgD);
      mtcPort1.receive(msgE);
    }
  }

  mtcPort1.send(msgE);

  alt{
    [] mtcPort1.receive(msgF) {
      setverdict(pass);
      mtcPort1.send(msgA);
    }
  }

  deactivate(alt_Fail_reference);

  mtcPort1.send(msgE);

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.10: An incorrectly refactored version of the Listing 6.9 using the Replace Altstep with Default *refactoring.*

This is a very peculiar example. As we see in Figure 6.15 (a) and (b), because both instances are running independently, during the third execution, at the first *alt* statement, the original test case (Listing 6.9) will select the right branch, with *port1.receive(msgC)*, because the left branch has been completely covered, whereas the incorrectly refactored one (Listing 6.10) will still select the left branch, with *port1.receive(msgB)*, because there are still some alternative paths below it that have not yet been covered. This will most likely result in the mismatching of all traces from that point on and disrupt all the remaining executions, even though some of them may still be able to match (Figure 6.16). This is not very useful if we want to localize the exact point where a difference in behavior was introduced. Furthermore, there could be other deeply nested discrepancies which may be hard to localize, if all the traces mismatch after a given point. Thus, we will have to come up with a better strategy for such cases.

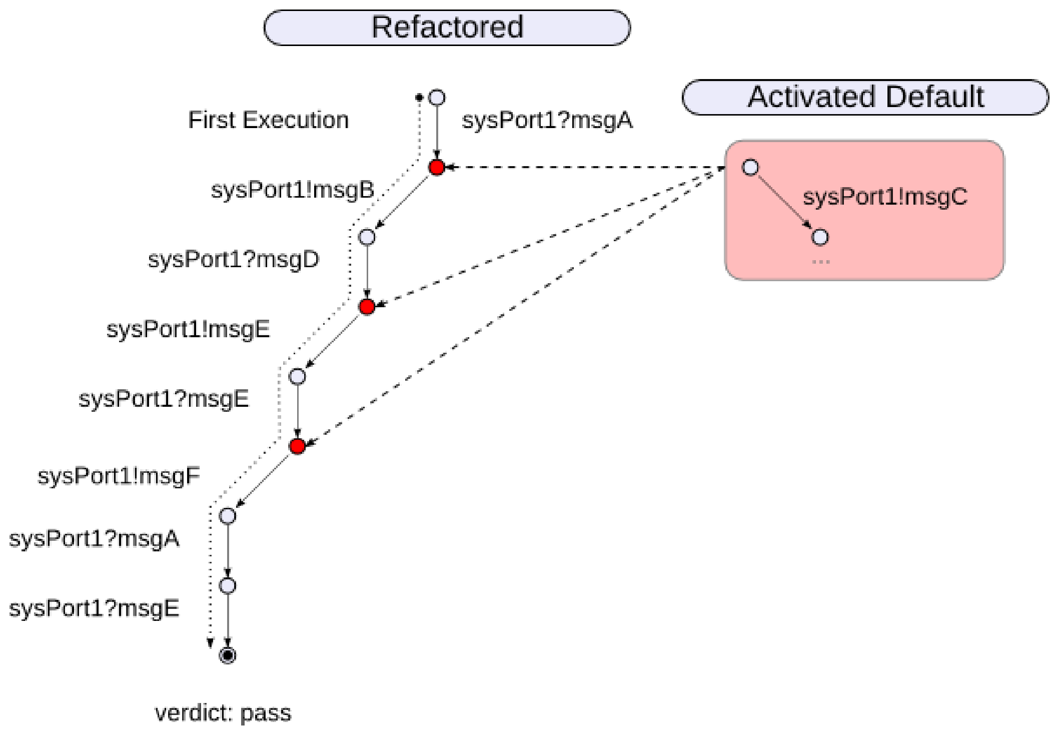

Figure 6.12: Intermediate behavior representation during the first execution of the incorrectly refactored test case using the Replace Altstep with Default *refactoring in Listing 6.10.*

The strategy is called *behavior alignment*. We mentioned in the requirements that the simulation may optionally be directed, i.e. one of the simulated SUT instances may surrender its control over the execution of the corresponding test case to the other simulated SUT that is executed simultaneously. This can be especially useful in the current case and similar ones.

The simulated SUT will surrender the control only temporary and suspend its backtracking until the additional behavioral paths of the simulated SUT that takes control are covered, and during that time span we hope to uncover the exact point where the extra behavior was introduced. After the additional paths have been covered, the behaviors of both simulated SUTs can be realigned back to taking the same branches automatically and they can continue to execute independently with backtracking resumed on the simulated SUT that had previously surrendered its control and continuing from the point it was previously suspended.

The simulated SUT that surrendered its control (in this case the one simulating the original test case) will follow the actions of the simulated SUT that is in control (the

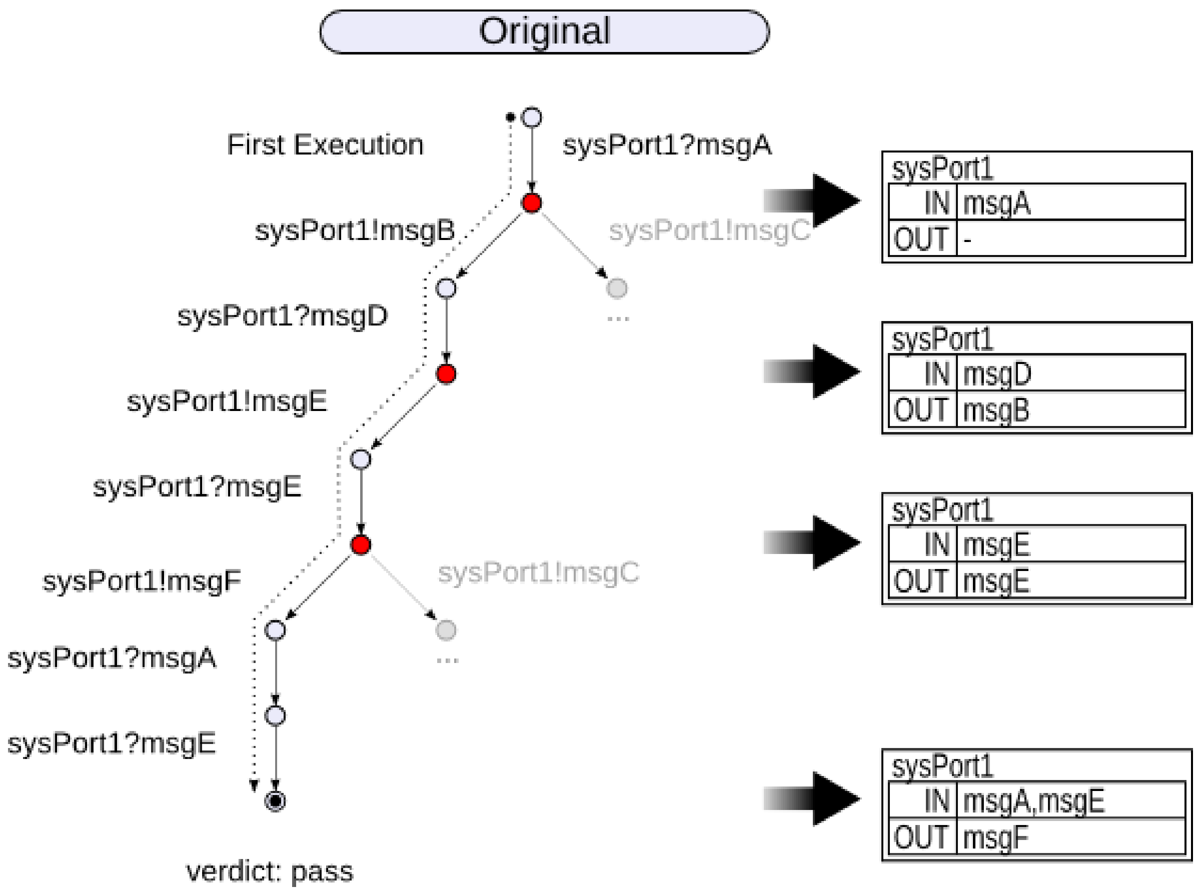

Figure 6.13: Behavior representation and comparison traces during the first execution of the original test case in Listing 6.9.

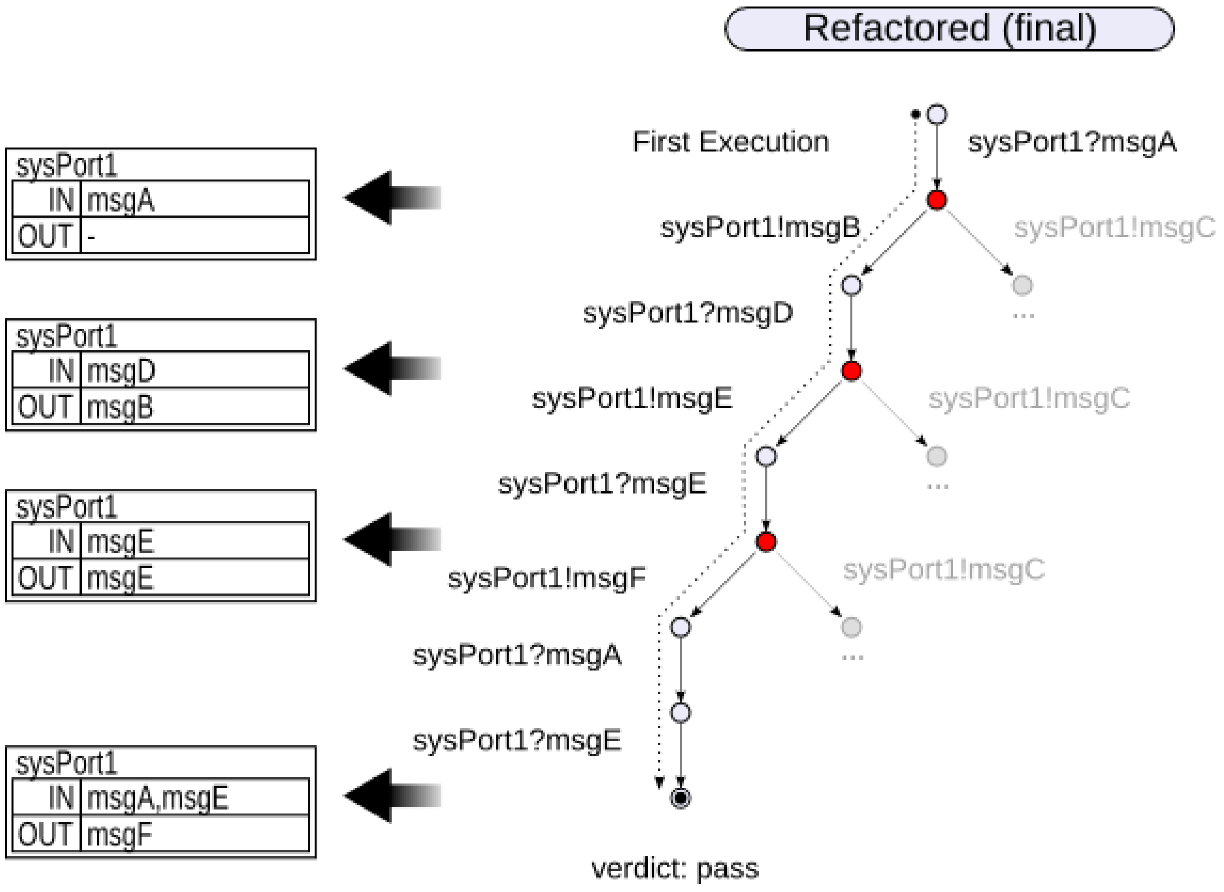

Figure 6.14: Actual behavior representation and comparison traces during the first execution of the incorrectly refactored test case using the Replace Altstep with Default *refactoring in Listing 6.10 constructed from Figure 6.12.*

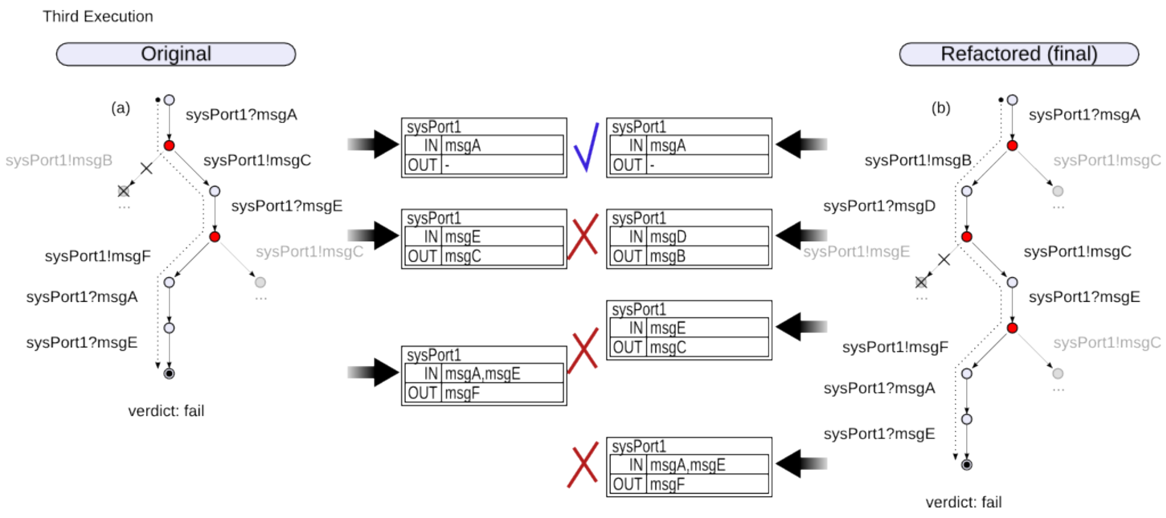

Figure 6.15: Behavior representation and comparison traces during the third execution of (a) the original test case in Listing 6.9 and (b) the incorrectly refactored test case using the Replace Altstep with Default *refactoring in Listing 6.10.*

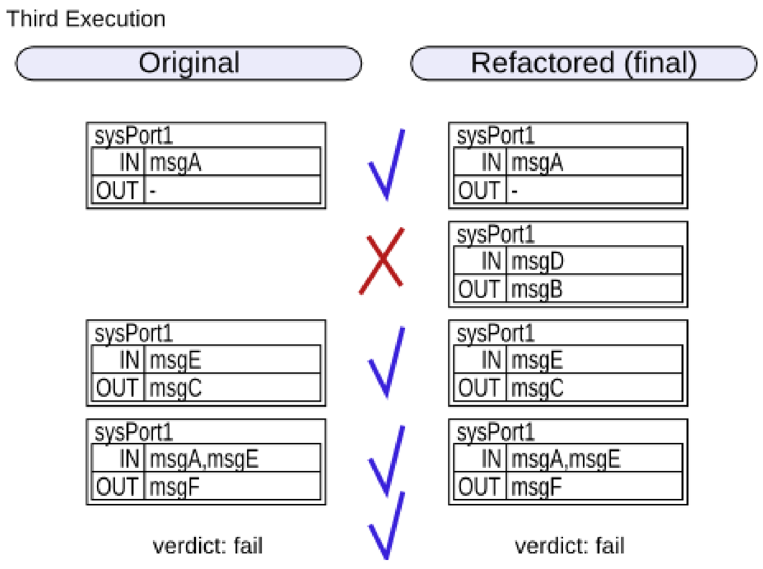

Figure 6.16: Suggested alignment of the traces generated during the third execution from the behavior representations in Figure 6.15.

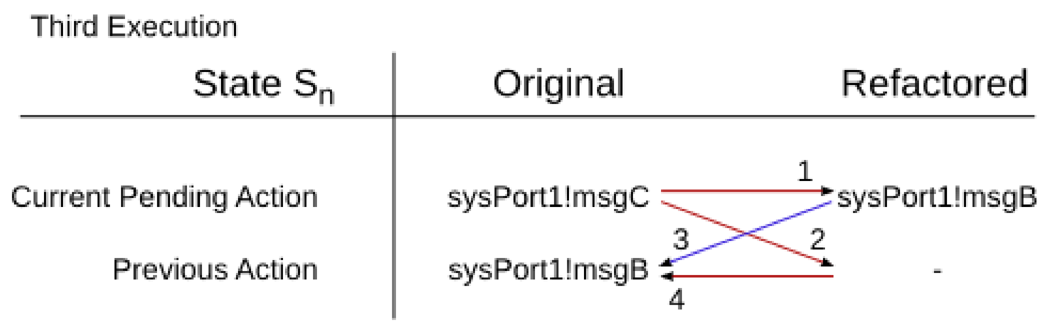

Figure 6.17: Cross-comparison of the pending actions for behavior alignment during the third execution of the original (Listing 6.9, Figure 6.15 (a)) and refactored (Listing 6.10, Figure 6.15 (b)) test cases.

one simulating the incorrectly refactored test case) whenever possible, and when there is no available action to match the one taken by the simulated SUT in control, it will simply take the last (or rightmost) available action at that point.

We will have to employ a lookahead tactic to identify the simulated SUT that has additional behavior and which has to surrender its control over the execution. After the comparison at the blocking point has been completed, the comparator will additionally look at the actions that are pending on each port. If the pending actions are all the same, then it can continue as normal. If they are not, we will compare the current pending actions of both simulated SUTs to the previous actions of the other simulated SUT at that point (following the precedence, right to left). If the previous pending action of a simulated SUT matches the current pending action of the other simulated SUT, then the former has to surrender its control to the later which has additional behavior under that branch. Consider the current case – the behavior models in Figures 6.15 – at the first *alt* statement, as already mentioned, during the third execution the original test case will have *port1.receive(msgC)* as its pending action, whereas the incorrectly refactored one will have *port1.receive(msgB)* as its pending action. From the SUT's perspective the pending actions are *sysPort1!msgC* and *sysPort1!msgC* respectively. The comparator will recognize this difference in the pending actions and request the previous actions available at that state from the simulator and then perform cross-comparison on the current pending actions and the previous ones from both simulated SUTs (Figure 6.17). The indexes represent the order of the comparison steps.

Figure 6.18 illustrates the alignment of simulation behaviors. They will be aligned upon every subsequent execution until both can select the same branch without intervention. The aligned traces are presented in Figure 6.19

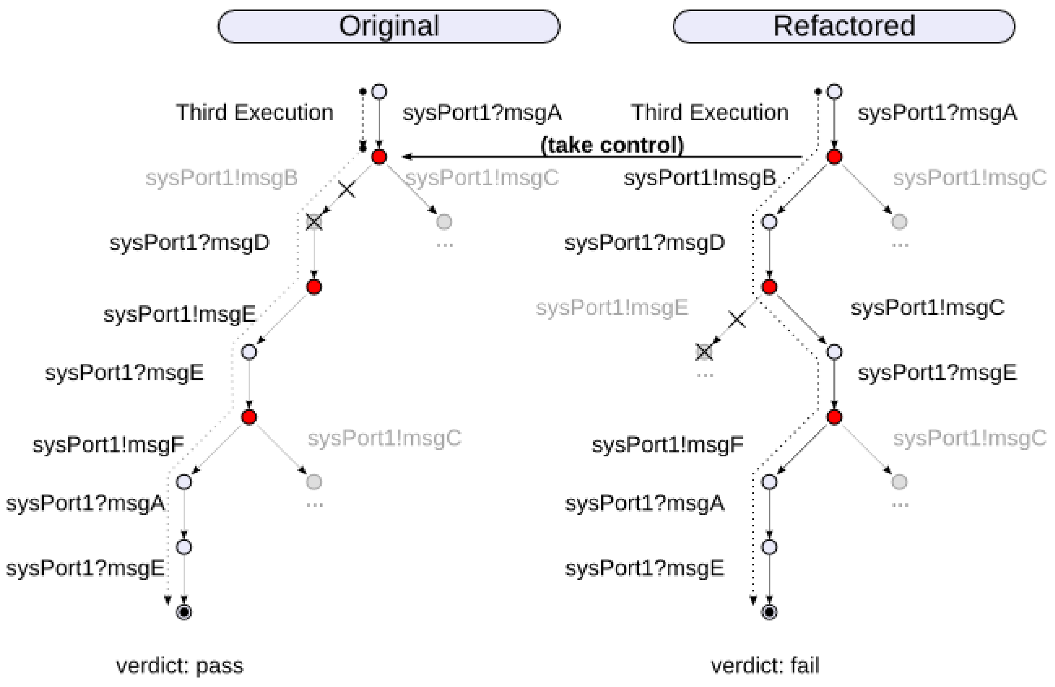

Figure 6.18: The aligned behaviors of the original and the incorrectly refactored test cases during their third execution.

If the cross-comparison of the current and previous pending actions does not find any matches, then we simply continue as normal, logging the observation and the traces from the current execution. There are generally two situations where the cross-comparison will fail – the pending actions are different – the messages to be sent are different (Figure 6.20), in which case at least step 4 may find that the previous pending actions from both simulated SUTs were equivalent, or there is a branch introduced or removed at that very point (Figure 6.21). Should either one of these be the case, the the whole equivalence process will be aborted at the end of the execution, as the results from that point on will be unreliable.

We are inspecting the pending actions on the ports and not the test components directly. This is for the same reason we are not comparing test components directly – they may not correspond to each other in both test systems.

We may have applied a similar concept of looking ahead to try to identify different or additional/missing branches upfront, before we get to select them. It may sound plausible, but we cannot predict directly how many executions it will take to select that branch and hence we will be facing the similar situation as well at some point so it will not have the desired effect of avoiding such complications.

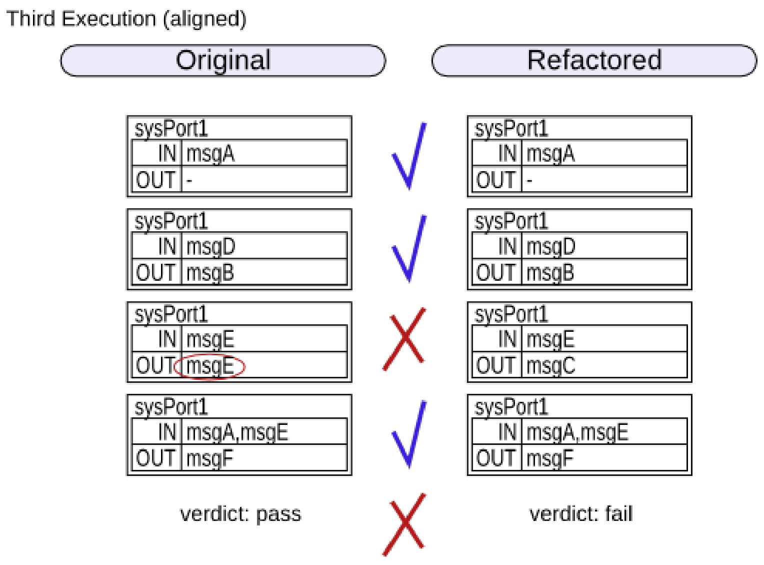

Figure 6.19: The aligned traces resulting from the alignment of the behaviors (Figure 6.18) during the third execution.

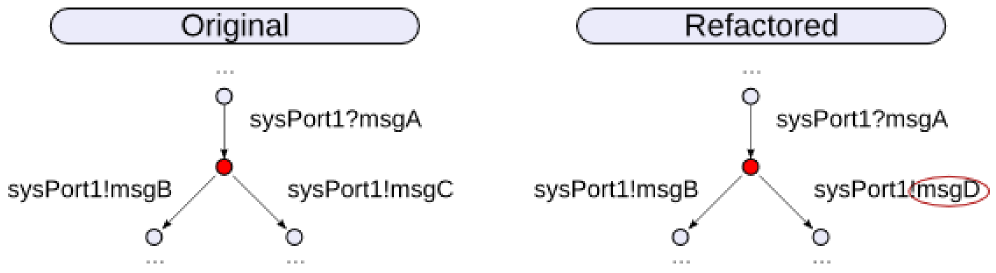

Figure 6.20: Cross-comparison failure cause - different actions.

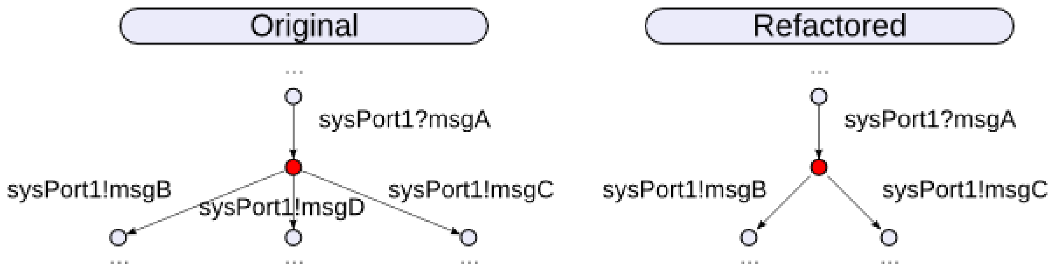

Figure 6.21: Cross-comparison failure cause - missing branch.

If messages that are sent from the test system are different, it does not affect the simulation critically. Therefore, such occurrences are logged, but no special action is required to handle them.

6.3 Replace Template with Modified Template

Synopsis: Templates of the same structured or list type with similar contents that differ in only a few fields, can be reduced to one full base template and other simplified templates that specify only the fields that are different from the base template with the help of the modifies keyword ([29], Section 4.2.9, p.55).

Possible Issues: The *Replace Template with Modified Template* refactoring is applied to complex data types, therefore one must proceed with caution while applying it. Failing to overwrite a field, overwriting one field more than necessary, or the wrong one may all introduce changes to the data being exchanged during testing and even affect the behavior of the test system, directly or indirectly. Furthermore, an error introduced to a modified template that is used to modify further templates may result in chain errors.

Application Examples: Consider the following example: We have a set of entities with properties, say phone numbers with several properties, as listed in Table 6.1.

Name	Prefix	Internal Line	Code	Internal
secretary	55	66	0	true
chief-direct	55	66	1	true
chief-mobile	55	99	1	false
maintenance	55	77	0	true
support	55	99	0	false

Table 6.1: Sample phone number database

The sample data can be represented using five templates in TTCN-3 (Listing 6.11). Consider it being the phone database of a small company's central department – 55 is the company phone line, 66 is the central department direct line, 77 is the maintenance department line and 99 are external lines. The codes are assigned to specific roles – 0 is the main entry point – the secretary's office, 1 connects to the bosses lines, internal or external (mobile). Note that 0 is also used for support and maintenance as these are the main entry points for the respective departments, where the support line is also external (probably outsourced to another country).

```
template phoneNumber secretary := {
  prefix := 55,
  internal_line := 66,
  code := 0,
  internal := true
}

template phoneNumber chief_direct := {
  prefix := 55,
  internal_line := 66,
  code := 1,
  internal := true
}

template phoneNumber chief_mobile := {
  prefix := 55,
  internal_line := 99,
  code := 1,
  internal := false
}

template phoneNumber maintenance := {
  prefix := 55,
  internal_line := 77,
  code := 0,
  internal := true
}

template phoneNumber support := {
  prefix := 55,
  internal_line := 99,
  code := 0,
  internal := false
}
```

Listing 6.11: TTCN-3 templates for the data in Table 6.1

```
template phoneNumber secretary := {
  prefix := 55,
  internal_line := 66,
  code := 0,
  internal := true
}

template phoneNumber chief_direct modifies secretary := {
  code := 1
}

template phoneNumber chief_mobile  modifies secretary := {
  internal_line := 99,
  code := 1,
  internal := false
}
```

```

template phoneNumber maintenance  modifies secretary := {
  internal_line := 77
}

template phoneNumber support  modifies secretary := {
  internal_line := 99,
  internal := false
}
```

Listing 6.12: The TTCN-3 templates from Listing 6.11 correctly refactored using the Replace Template with Modified Template refactoring

Listing 6.12 illustrates the correct application of the *Replace Template with Modified Template* refactoring to our sample data. Validating that this was indeed a correct application of the refactoring using our approach will be rather trivial, given a test case that uses the refactored templates. We will be more interested in validating an incorrect application of the refactoring and proving that the resulting test case is not equivalent to the original. Given the incorrectly refactored templates in Listing 6.13 and the sample test case in Listing 6.14, we will analyze the two types of changes to behavior that will result from the two errors we introduced.

```
template phoneNumber secretary := {
  prefix := 55,
  internal_line := 66,
  code := 0,
  internal := true
}

template phoneNumber chief_direct modifies secretary := {
  code := 1
}

//code field is not modified as originally intended to be 1,
//thus calling this version of the chief_mobile will land on the support line
template phoneNumber chief_mobile  modifies secretary := {
  internal_line := 99,
  internal := false

}

template phoneNumber maintenance  modifies secretary := {
  internal_line := 77
}

//internal field is not modified as originally intended to be false,
//calling the support like this will fail to include the charge notification
template phoneNumber support  modifies secretary := {
  internal_line := 99
}
```

Listing 6.13: The TTCN-3 templates from Listing 6.11 incorrectly refactored using the Replace Template with Modified Template refactoring

```
testcase pbx_test1(in phoneNumber number) runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  timer t;

  //send a request for call to a number
  mtcPort1.send(number);

  //if the number is not internal, the PBX system should send a
  //charge notification and await confirmation
  if ( phoneNumber.internal == false){
    t.start(5.0)
    alt{
      [] mtcPort1.receive(chargeNotification) {
        setverdict(pass);
        mtcPort1.send(chargeConfirmation);
      }
      [] t.timeout {
        setverdict(fail);
      }
    }
  }

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.14: A sample test case using the phone number templates as parameters

Running the test case with the two incorrectly refactored templates will result in two types of changes to behavior. Using the incorrectly refactored *chief_mobile* template as a parameter for the test case would produce the same behavior representation as the original and superficially there is no noticeable difference. When comparing the traces however, our comparator will be able to spot the difference in the complex data structure which will be reflected in the encoded representation of the data.

Much more interesting is the situation when we use the incorrectly refactored support template. The changes in the data have a direct impact on the behavior of the test system. Since charge confirmation will be expected only for external calls (if the template used has its internal field set to false), our simulator will not enter the if scope for the refactored test case, resulting in a different behavior representation and subsequently missing traces for comparison.

6.4 Parametrize Template

Synopsis: Multiple templates of the same type with different values for the same fields can be replaced with a single parametrized template ([29], Section 4.2.10, p.57).

Possible Issues: Similar issues as with the *Replace Template with Modified Template* refactoring can occur, namely overlooking an additional differing field.

Application Examples: Validating a correct application of the refactoring should again be rather trivial, therefore we will be more concerned with incorrect applications of the *Parametrize Template* refactoring. We will consider the PBX samples illustrated in Listing 6.11 again. This time, instead of replacing the templates with modified templates, we will try to parametrize them. We will use two parametrized templates - one for internal and one for external calls (Listing 6.15).

```
//incorrectly assumed that all internal numbers have the same internal line
template phoneNumber call_internal (integer call_code) := {
  prefix := 55,
  internal_line := 66,
  code := call_code,
  internal := true
}

template phoneNumber call_external (integer call_code) := {
  prefix := 55,
  internal_line := 99,
  code := call_code,
  internal := false
}
```

Listing 6.15: An unsuccessful attempt to refactor the templates from Listing 6.11 using the Parametrize Template refactoring

The assumption in this case is that all internal numbers have the same internal line - 66, which is incorrect and resulted probably from overlooking the difference in the *internal_line* field in the original *maintenance* template. We can therefore easily notice that the *maintenance* phone template cannot be correctly substituted by the *call_internal()* parametrized template, because the value of the *internal_line* field is different. Attempting to do so will be recognized by our approach as a mismatching data exchange type of error. Similarly to the *Replace Template with Modified Template* refactoring, the incorrect application of the Parametrize Template may lead to even more drastic changes to the test case behavior in other contexts.

6.5 Prefix Imported Declarations

Synopsis: When referencing imported declarations, prefixes can be used to avoid name clashes. ([29], Section 4.2.16, p.69)

Possible Issues: Incorrect application of the Prefix Imported Declarations refactoring to a reference of a local variable may cause incorrect data to be exchanged or even more drastic changes to behavior, similar to the changes illustrated in the incorrect application of the *Replace Template with Modified Template.*

Application Examples: First, we would like to address a controversial example from [29] (Section 4.2.16, p.70). Listings 6.16 and 6.17 illustrate the situation.

```
module ExampleModule {
  group types {
    type charstring SipMethod ("REGISTER", "INVITE", "ACK", "BYE",
      "CANCEL", "OPTIONS");
  }
}

module DependantModule {
  import from ExampleModule all;

  type charstring SipMethod;

  testcase tc_exampleTestCase() runs on ExampleComponent {
    // ...
    var SipMethod v_method := "QUIT";
  }

}
```

Listing 6.16: The original version as presented in Listing 4.33 in [29].

```
module ExampleModule {
  group types {
    type charstring SipMethod ("REGISTER", "INVITE", "ACK", "BYE",
      "CANCEL", "OPTIONS");
  }
}

module DependantModule {
  import from ExampleModule all;

  type charstring SipMethod;

  testcase tc_exampleTestCase() runs on ExampleComponent {
    // ...
    var SipMethod v_method := "QUIT";
  }

}
```

Listing 6.17: The refactored version as presented in Listing 4.33 in [29].

This example illustrates possible problems with the *Prefix Imported Declarations* refactoring. The problem is that the referenced and subsequently prefixed *v_method* variable cannot be of the (sub-) type *SipMethod* from the *ExampleModule*, because the value assigned to it ("*QUIT*") is not in the restricted list defined in *ExampleModule*.

Although, this specific example causes a runtime error, there are more subtle cases where such incorrect prefixing of imported declarations may not be immediately obvious.

There are two possible usage scenarios of the module prefixing – to make the references to imported declarations more obvious, which can improve the readability and maintainability of a test case, and to reference imported declarations that have been "*shadowed*" (overridden by local declarations in the current scope). Only the first can be considered refactoring. Should the second one become necessary, it is no longer considered a refactoring. In the first usage scenario however, in the course of application of module prefixing, references to local declarations that are supposed to shadow the imported ones, may also get prefixes, effectively making them references to the imported declarations as well, which may change the observable behavior of a test case. Additionally, if there is a name clash in the original test case, applying the *Prefix Imported Declarations* to fix it, may result in an intentional change to behavior, which already defies the purpose of refactoring and the question arises which version, the original or the refactored one, has the correct behavior. We are more concerned with situations where changes to the behavior are not intentional.

Constructing a meaningful example to illustrate correct usage of the *Prefix Imported Declarations* refactoring proves to be a non-trivial task. We will present an example, which by no means qualifies as properly designed. Listing 6.18 presents us with the base situation. We have two modules – *PBX_revised* and *PBX_imported*. Only the relevant parts of them are presented. [3]

```
1   module PBX_imported {
2     //...
3     const integer external_code := 99;
4     const integer main_department_code := 66;
5     const integer maintenance_department_code := 77;
6     const integer support_department_code := external_code;
7     //...
8   }
9
10  module PBX_revised {
11    import from PBX_imported all;
12    //...
13    testcase pbx_test2(in phoneNumber number) runs on mtcType system systemType{
14      map(mtc:mtcPort1, system:systemPort1);
15
16      timer t;
17
18      //send a request for call to a number
19      mtcPort1.send(number);
20
21      //instead of the boolean 'internal' flag, we will use
22      //the internal_line value
23      //external_code here references the imported external_code
24      if ( phoneNumber.internal_line == external_code){
25        t.start(5.0)
26        alt {
27          [] mtcPort1.receive(chargeNotification) {
```

[3]Lines 37–51 may need to be extracted as a function (by applying the *Extract Function* refactoring), since some TTCN-3 tools generate a runtime error, if the imported constant is referenced without a module prefix and there is a local variable with the same name declared later in the same scope.

```
          setverdict(pass);
          mtcPort1.send(chargeConfirmation);
        }
        [] t.timeout {
          setverdict(fail);
        }
      }
    }

    var integer external_code := 98;

    //external_code here references the freshly defined variable == 98
    if ( phoneNumber.internal_line == external_code){
      t.start(5.0)
      alt{
        [] mtcPort1.receive(chargeNotification) {
          setverdict(pass);
          mtcPort1.send(chargeConfirmation);
        }
        [] t.timeout {
          setverdict(fail);
        }
      }
    }
}
```

Listing 6.18: A modified version of the pbx_test1 *test case using definitions from an imported module.*

The test case *pbx_test2* is a modification of the *pbx_test1* (Listing 6.14), that checks the *internal_line* field instead of the *internal* boolean flag. If it corresponds to the value of the *external_code* ('99') in the *PBX_imported* module (assuming that *PBX_revised* does not define its own *external_code* constant) then confirmation will be requested. Now supposedly an inexperienced test implementor wanted to quickly include the the value '98' as a code for an external line as well, so they simply copied the segment and introduced an overriding variable with the same name, but a different value. The test case now checks whether a charge notification is received when a call is requested to a number which has either '99' or '98' as its *internal_line* field.

```
testcase pbx_test2_refactored (in phoneNumber number)
  runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  timer t;

  //send a request for call to a number
  mtcPort1.send(number);

  //instead of the boolean 'internal' flag, we will use
  //the internal_line value
  //external_code here references the imported external_code
  //and it is made obvious by using the prefix
  if ( phoneNumber.internal_line == PBX_imported.external_code){
    t.start(5.0)
    alt{
      [] mtcPort1.receive(chargeNotification) {
        setverdict(pass);
        mtcPort1.send(chargeConfirmation);
      }
      [] t.timeout {
        setverdict(fail);
      }
    }
  }

  var integer external_code := 98;

  //external_code here references the freshly defined variable == 98
  if ( phoneNumber.internal_line == external_code){
    t.start(5.0)
    alt{
      [] mtcPort1.receive(chargeNotification) {
        setverdict(pass);
        mtcPort1.send(chargeConfirmation);
      }
      [] t.timeout {
        setverdict(fail);
      }
    }
  }

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.19: A correctly refactored version of the pbx_test2 *(Listing 6.18) test case using the* Prefix Imported Declarations *refactoring.*

Listing 6.19 presents the correct application of the *Prefix Imported Declarations* refactoring. Applying our approach to validate that its behavior is equivalent to that of the original test case will be trivial. Listing 6.20 on the other hand illustrates an incorrect application of the *Prefix Imported Declarations* refactoring, where both references of the *external_code* were prefixed and as a result the test case has dramatically changed its behavior in that it will expect two charge notifications if the *internal_line* field of the number has the value of '99' and will also not expect any charge notification if the value of that field is '98'. Such changes in behavior will also be reflected in the traces for comparison.

```
testcase pbx_test2_refactored_v2 (in phoneNumber number)
  runs on mtcType system systemType{
  map(mtc:mtcPort1, system:systemPort1);

  timer t;

  //send a request for call to a number
  mtcPort1.send(number);

  //instead of the boolean 'internal' flag, we will use
  //the internal_line value
  //external_code here references the imported external_code
  if ( phoneNumber.internal_line == PBX_imported.external_code){
    t.start(5.0)
    alt{
      [] mtcPort1.receive(chargeNotification) {
        setverdict(pass);
        mtcPort1.send(chargeConfirmation);
      }
      [] t.timeout {
        setverdict(fail);
      }
    }
  }

  var integer external_code := 98;

  //external_code here should references the freshly defined variable == 98
  if ( phoneNumber.internal_line == PBX_imported.external_code){
    t.start(5.0)
    alt{
      [] mtcPort1.receive(chargeNotification) {
        setverdict(pass);
        mtcPort1.send(chargeConfirmation);
      }
      [] t.timeout {
        setverdict(fail);
      }
    }
  }

  unmap(mtc:mtcPort1, system:systemPort1);
}
```

Listing 6.20: An incorrectly refactored version of the pbx_test2 *(Listing 6.18) test case using the* Prefix Imported Declarations *refactoring.*

In other contexts it may again be the case that an incorrect application of this refactoring may not be visible from the outside and thus our approach will not be able to identify the changes to behavior, because they cannot be externally observed and thus make no difference.

6.6 Extract Function

Synopsis: Repeated code fragments, long functions or test cases can all be moved to a new function to improve reusability, readability and maintainability and in turn reduce code duplication ([29], Section 4.1.2, p.36).

Possible Issues: Similar to the *Extract Altstep* refactoring, we are confronted with possible issues related to parameter passing and neglecting variables and scope.

Application Examples: We would only examine briefly another example from Zeiss' original work on refactorings in TTCN-3 [29]. Consider Listings 6.21 and 6.22:

```
module ExtractFunctionExample {
  // ...
  type component ExampleComponent {
    timer t;
    port ExamplePort pt;
  }
    // ...
  function f_sendMessages(in float p_duration)
    runs on ExampleComponent {

    timer t;
      t.start( p_duration );
      t.timeout;

    pt.send( a_MessageOne );

    t.start( p_duration );
    t.timeout;

    pt.send( a_MessageTwo );
  }
}
```

Listing 6.21: The original version as presented in Listing 4.1 in [29].

```
module ExtractFunctionExample {
    // ...
  type component ExampleComponent {
    timer t;
    port ExamplePort pt;
  }
  // ...
    function f_wait(in float p_duration)
      runs on ExampleComponent{
      t.start( p_duration );
      t.timeout;
    }

    function f_sendMessages(in float p_duration)
      runs on ExampleComponent {
      timer t;

      f_wait(p_duration);
      pt.send( a_MessageOne );

      f_wait(p_duration);
      pt.send( a_MessageTwo );
    }
}
```

Listing 6.22: The refactored version as presented in Listing 4.2 in [29].

This is another controversial example[4]. We may argue that the differences in the behavior are yet subtle, but may have serious effect on the observable behavior. The problem is that there is a new timer defined within the scope of the *f_sendMessages* function (Listing 6.21, Line 14), and we may observe that it "*shadows*" the component timer definition (Listing 6.21, Line 5) within the scope of the function. However, *shadowing* in this context is not possible. The timers, variables, and constants declared within test component type definitions are visible to test cases, functions, and altsteps that are declared specific to the component type with a **runs on** clause. Therefore they cannot be declared again within such a function. Attempting to do so will therefore result in a compilation error. If we choose to remove the component timer declaration, then the extracted function will cause a compilation error since the timer reference in its scope is undefined. It will be necessary to pass it as a parameter. If we choose to remove the timer declaration in *f_sendMessages*, then the function will be correct and compile without an error. The extracted function will also work correctly. The argument in [29] also suggests that the second timer declaration within *f_sendMessages* was merely overseen.

This concludes our studies on the applicability of our approach to a few sample cases.

[4]This example features only a function, not a distinct test case, but it may very well be used in a test case and therefore affect its behavior.

7 Summary and Outlook

Refactoring is essential for the maintenance of the internal structure of TTCN-3 test cases, for the improvement of their internal quality and integrity, but it is only applicable if the quality of the refactoring process itself and its results can be validated.

The primary objective of this project was the validation of refactored test cases. The most significant property of refactorings is preservation of observable behavior. If it does not hold, the refactoring was not applied successfully. Our ultimate goal was to design an applicable approach for the equivalence checking of observable test case behavior.

We presented some background information on key concepts we are building upon, such as the language of interest – TTCN-3, refactoring in general and in the context of TTCN-3, distributed systems, and bisimulation. Some problems associated with these were also identified. Then, we established key concepts in the context of this thesis, such as the notion of observable behavior are concerned with, the representation of the observable behavior, and what is considered equivalent behavior.

The equivalence checking approach presented in this thesis is based on bisimulation. It is comprised of several conceptual entities – an execution environment, a comparator and a logger. The execution environment is based on existing tools for the execution of TTCN-3 test cases and a self-configuring simulated SUT, that guides the test case execution into exhaustively covering all behaviors that can externally induced and observed. The comparator uses traces provided by two simultaneously running execution environments that execute two test cases, to check the test cases for equivalent observable behavior on the fly. The logger provides clues for the identification and localization of errors in the application of refactorings that introduce changes to the observable behavior and for post analysis. Essentially, the concept of performing tasks on the fly was main design principle throughout this thesis, particularly as means to manage the state space explosion problem. Therefore, all the conceptual entities in our approach had to comply to this principle. The entities themselves are designed for standalone operation and can be used for other purposes as well.

The important aspects of our prototypical implementation and other implementation specific issues were briefly presented, followed by some examples were provided to illustrate and study the applicability of our approach and some interesting insights came out.

As the examples have shown, things can go wrong when refactorings are applied. Incorrect application of refactorings can affect the behavior of TTCN-3 test cases. Therefore, a proof, that the observable test case behavior is preserved, is necessary, even after the application of rather trivial refactorings. The approach presented in this thesis has been designed specifically for this purpose. The examples we used to illustrate its applicability, have shown that the approach can be effective in checking whether the observable behavior of a test case has been preserved or not. Its effectiveness remains to be studied using larger data sets. We have analyzed only a limited set of refactorings. Analysis of further refactorings may provide further insights.

But one of the most significant observations is that, if a change to behavior does not manifest itself in the observable behavior of the test case at its current development stage, when it is checked for equivalence, it will not be detected by our approach. Such changes may manifest themselves at later development stages, when the test case is extended or modified further, and only then we will face the effects of the incorrect application of refactorings. Our approach will not be useful in such a scenario, unless both the original and the refactored version are further developed in the same direction (which is seldom the case), so that the original can be used as a basis for equivalence checking at later points. Therefore, we can conclude that our approach is only appropriate for detecting immediate changes to the test case behavior, which can be directly observed in the current version of a test case, right after it has undergone refactoring.

Concerning the future of our approach, extending the coverage of our approach to manage advanced aspects of TTCN-3 is a major point. Further studies with other refactorings and eventually real life examples to gain further insights is also an important point in the future developments of this project. This will first require collection of data for such purposes, which is a non-trivial task.

Partial and in-place equivalence checking may be an interesting feature. Based on it, an integration with refactoring tools to provide immediate feedback on the preservation of observable behavior will be an interesting prospect. With the help of in-place equivalence checking, a unified process for the application and validation of refactorings can be designed.

The individual usage of the separate tools can span to other application areas. Simulation could be useful for tryouts, teaching, self-learning, or conducting other experiments, such as checking whether a certain property holds (e.g. a test verdict is set in all possible executions). Comparison could be used, to monitor performance of different tools and environments. Logging could supplement the test logging interface of the TTCN-3 test system, by providing additional customizable information at runtime.

A replay feature may be of some interest. Replaying or reproducing the executions that lead to errors can further facilitate quick validation that a particular problem was corrected, followed by a complete revalidation to check if other problems were introduced with the supposed corrections.

The notion of equivalence is always a questionable issue. It is generally defined for a specific context. Our notion of equivalence may need to be redefined with further developments of the approach to suit evolving needs. Adding timing constraints may be one of the critical improvements. The partial ordering of sequential events may also play a role in some contexts such as testing shared media. Thus, this concept may need to be investigated further and eventually integrated in our equivalence checking approach, which will also require a revision of our notion of equivalence.

A possible use for the purposes of the *European Telecommunications Standard Institute* (ETSI) Specialist Task Force 343 (STF 343) [3] for the validation of TTCN-3 tools has also been briefly discussed. In such a case we will need to simply use tools from different vendors for the two test systems and execute the same test cases on both test systems instead of using the same test systems running different versions of a test cases.

Even though this project has been designed around a specific execution environment, because TTCN-3 is a standardized language and the tools therefore have to implement these standards, and because of its modular and minimally intrusive architecture, our approach should be easily portable to other execution environments as well. Due to the unavailability of TTCN-3 tools from other vendors at the time, it remains to be seen whether this would really be the case.

Abbreviations and Acronyms

CD Codec

CH Component Handling

ETSI *European Telecommunications Standard Institute*

FIFO First-In-First-Out

FSA Finite State Automaton

IDE *Integrated Development Environment*

LTS Labeled Transition System

MTC *Main Test Component*

MSC Message Sequence Chart

PA Platform Adapter

PCOs Points of Control and Observation

PTCs *Parallel Test Components*

RMI Remote Method Invocation

SA System Adapter

STF 343 Specialist Task Force 343

SUT *System Under Test*

TCI Test Control Interface

TE Test Executable

TL Test Logging

TM Test Management

TRI Test Runtime Interface

TSI Test System Interface

TRex *TTCN-3 Refactoring and Metrics Tool*

TTCN-3 *Testing and Test Control Notation Version 3*

UM Universal Message

List of Figures

2.1 Standard TTCN-3 test system architecture 6
2.2 A sample TTCN-3 test configuration 9

4.1 LTS representation of the sample TTCN-3 test behavior in Listing 4.1 22
4.2 Derivation of the corresponding SUT behavior representation from the test system behavior specified in Listing 4.1 23
4.3 Sample test configuration for Listing 4.1 with component1.port1 mapped to system.sysPort1 24
4.4 The test system behavior specified specified in Listing 4.1 from the SUT's perspective with test configuration (from Figure 4.3) taken into account . . . 24
4.5 The test system behavior specified specified in Listing 4.1 from the SUT's perspective using standard telecommunications notation 25
4.6 Conceptual illustration of equivalence checking on the fly 28
4.7 Simplified MSC illustrating the use of instrumentation at runtime 31
4.8 Handling of default altsteps during test case execution 33
4.9 Complete behavior representation of the test case fragment in Listing 4.3 . . 37
4.10 Behavior coverage during the first execution of the test case fragment in Listing 4.3 (a) and marking nodes as "checked" during the backtracking phase (b) 37
4.11 Behavior coverage during the second execution of the test case fragment in Listing 4.3 (a) and marking nodes as "checked" during the backtracking phase (b) 38
4.12 Behavior coverage during the third execution of the test case fragment in Listing 4.3 (a) and marking nodes as "checked" during the backtracking phase (b) 38
4.13 Reduction of the behavior models during backtracking after the first execution 40
4.14 Reduction of the behavior models during backtracking after the second execution 40
4.15 Reduced versions of the behavior models from Figure 4.11 after applying backtracking with reduction 41
4.16 Reduced versions of the behavior models from Figure 4.12 after applying backtracking with reduction 41

4.17 Sample full behavior representations for the two last initialized test components 42
4.18 Illustration of the token-passing algorithm applied to the sample behaviors in Figure 4.17 during after the first (a), second (b1,b2) and third (c) executions 43
4.19 TSI conceptual hierarchy (a) and our compact representation to reflect this hierarchy (b) 45
4.20 Comparison of TSI structures 46
4.21 Tracing process 46
4.22 Equivalence checking process workflow summary 49
4.23 Conceptual architecture of the equivalence checking approach 49

5.1 MSC illustrating the data flow during a send operation in TTCN-3 52
5.2 MSC illustrating the data flow during a receive operation in TTCN-3 52
5.3 MSC illustrating the usage of UMs to circumvent decoding in our approach . 54
5.4 Implementation architecture of our approach 57
5.5 Our approach in the context of the TTCN-3 standard architecture 58

6.1 Behavior representation and traces for comparison during the first execution of (a) the original test case as specified in Listing 6.1, and (b) the refactored test case as specified in Listing 6.2. 63
6.2 Behavior representation and traces for comparison during the second execution of (a) the original test case as specified in Listing 6.1, and (b) the refactored test case as specified in Listing 6.2. 63
6.3 Comparison of the traces generated during the third execution of the original and refactored test cases in Listings 6.1 and 6.2 respectively. 65
6.4 Comparison of the traces generated during the fourth execution of the original and refactored test cases in Listings 6.1 and 6.2 respectively. 65
6.5 Behavior representation and traces for comparison during the second execution of (a) the modified original test case as specified in Listing 6.3, and (b) the incorrectly refactored test case using the incorrectly extracted altstep in Listing 6.4. 67
6.6 Behavior representation and traces for comparison during the fourth execution of (a) the modified original test case as specified in Listing 6.3, and (b) the incorrectly refactored test case using the incorrectly extracted altstep in Listing 6.4. 67
6.7 Intermediate behavior representation during the first execution of the correctly refactored test case using the *Replace Altstep with Default* refactoring in Listing 6.8. 71
6.8 Final behavior representation and comparison traces during the first execution of the correctly refactored test case using the *Replace Altstep with Default* refactoring in Listing 6.8 constructed from Figure 6.7. 71

6.9 Comparison of the traces generated during the second execution of the test cases refactored using the *Extract Altstep* refactoring (Listing 6.2) and subsequently *Replace Altstep with Default* refactoring (Listing 6.8) refactorings on the original test case in Listing 6.1. 72
6.10 Comparison of the traces generated during the third execution of the test cases refactored using the *Extract Altstep* refactoring (Listing 6.2) and subsequently *Replace Altstep with Default* (Listing 6.8) refactorings on the original test case in Listing 6.1. 72
6.11 Comparison of the traces generated during the fourth execution of the test cases refactored using the *Extract Altstep* refactoring (Listing 6.2) and subsequently *Replace Altstep with Default* (Listing 6.8) refactorings on the original test case in Listing 6.1. 73
6.12 Intermediate behavior representation during the first execution of the incorrectly refactored test case using the *Replace Altstep with Default* refactoring in Listing 6.10. 75
6.13 Behavior representation and comparison traces during the first execution of the original test case in Listing 6.9. 76
6.14 Actual behavior representation and comparison traces during the first execution of the incorrectly refactored test case using the *Replace Altstep with Default* refactoring in Listing 6.10 constructed from Figure 6.12. 77
6.15 Behavior representation and comparison traces during the third execution of (a) the original test case in Listing 6.9 and (b) the incorrectly refactored test case using the *Replace Altstep with Default* refactoring in Listing 6.10. 78
6.16 Suggested alignment of the traces generated during the third execution from the behavior representations in Figure 6.15. 78
6.17 Cross-comparison of the pending actions for behavior alignment during the third execution of the original (Listing 6.9, Figure 6.15 (a)) and refactored (Listing 6.10, Figure 6.15 (b)) test cases. 79
6.18 Aligned behaviors 80
6.19 Aligned traces 81
6.20 Cross-comparison failure cause - different actions. 81
6.21 Cross-comparison failure cause - missing branch. 81

Bibliography

[1] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, New York, NY, USA, 2007.

[2] M. L. Cornélio. *Refactorings as Formal Refinements.* PhD thesis, Universidade Federal de Pernambuco, Brasil, Mar. 2004.

[3] ETSI. ETSI Terms of Reference for Specialist Task Force STF 343 - (TC MTS) on "3rd Generation Mobile Reference Test System to enable TTCN-3 Tool Assurance". `http://portal.etsi.org/stfs/ToR/ToR343v32_MTS_TTCN-3_tool_assurance_3Gmobile.doc`.

[4] ETSI. European Standard (ES) 201 873-1 V3.4.1 (2005-08): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140, 2008.

[5] ETSI. European Standard (ES) 201 873-4 V3.4.1 (2005-08): The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140, 2008.

[6] ETSI. European Standard (ES) 201 873-5 V3.3.1 (2005-08): The Testing and Test Control Notation version 3; Part 5: TTCN-3 Test Runtime Interface. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140, 2008.

[7] ETSI. European Standard (ES) 201 873-6 V3.4.1 (2005-08): The Testing and Test Control Notation version 3; Part 6: TTCN-3 Test Control Interface. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140, 2008.

[8] C. J. Fidge. Partial orders for parallel debugging. *SIGPLAN Not.*, 24(1):183–194, 1989.

[9] M. Fowler. Refactoring Tools Section on the Refactoring Home Page. `http://www.refactoring.com/tools.html`.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.

[12] P. Makedonski. Code Instrumentation for the Equivalence Checking of TTCN-3 Test Case Behavior. Technical report, Institute for Computer Science, Georg-August-University of Göttingen, 2008.

[13] T. Mens. A Formal Foundation for Object-Oriented Software Evolution. In *Proc. Int. Conf. Software Maintenance*, pages 549–552. IEEE Computer Society Press, 2001.

[14] T. Mens, S. Demeyer, B. D. Bois, H. Stenten, and P. V. Gorp. Refactoring: Current research and future trends. *Electr. Notes Theor. Comput. Sci*, 82(3), 2003.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science (LNCS).* Springer, 1980.

[16] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[17] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[18] H. Neukirchen. Re-Usability in Testing. Presentation, TAROT Summer School 2005, June 2005.

[19] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[20] D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science (LNCS)*, pages 167–183. Springer, 1981.

[21] D. L. Parnas. Software Aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[22] J. Philipps and B. Rumpe. Roots of Refactoring. In K. Baclavski and H. Kilov, editors, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15, 2001.* Northeastern University, 2001.

[23] D. B. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois, 1999.

[24] G. Snelting and F. Tip. Reengineering Class Hierarchies Using Concept Analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.

[25] K. Tai and R. H. Carver. Testing of Distributed Programs. In A. Zomaya, editor, *Handbook of Parallel and Distributed Computing*, chapter 33, pages 955–978. McGraw-Hill, New York (USA), 1996.

[26] L. A. Tokuda. *Evolving Object-Oriented Designs with Refactorings.* PhD thesis, University of Texas at Austin, 1999.

[27] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring Test Code. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, May 2001.

[28] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3.* John Wiley & Sons, Ltd, 2005.

[29] B. Zeiss. A Refactoring Tool for TTCN-3. Master's thesis, Institute for Informatics, ZFI-BM-2006-05, ISSN 1612-6793, Center for Informatics, University of Göttingen, Mar. 2006.

[30] B. Zeiss, H. Neukirchen, J. Grabowski, D. Evans, and P. Baker. Refactoring and Metrics for TTCN-3 Test Suites. In *System Analysis and Modeling: Language Profiles. 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Revised Selected Papers. Lecture Notes in Computer Science (LNCS) 4320. DOI: 10.1007/11951148_10*, pages 148–165. Springer, Dec. 2006.

All URLs have been verified on November 8, 2008.