Master's Thesis

Automating Test Case Generation from Transition Systems via Symbolic Execution and SAT Solving

Elisabeth Jöbstl[1]

Institute for Software Technology (IST)
Graz University of Technology
A-8010 Graz, Austria

Advisor:	Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Co-Advisor:	Ass.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig

Graz, September 2009

[1] E-mail: elisabeth.joebstl@student.tugraz.at

Masterarbeit

Automatisierung der Testfallgenerierung aus Transitionssystemen mittels symbolischer Ausführung und SAT-Solving

Elisabeth Jöbstl[1]

Institut für Softwaretechnologie (IST)
Technische Universität Graz
A-8010 Graz

Gutachter: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Mitbetreuer: Ass.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig

Graz, im September 2009

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: elisabeth.joebstl@student.tugraz.at

Abstract

Testing is a crucial as well as labour-intensive task in software development. Automation can be applied to all phases of testing in order to improve its efficiency, to lower its costs, and hence to make it more viable. This work addresses the automation of the test case generation phase.

Nowadays, many model-based test case generation tools exist. However, most of them enumerate the specification's state space and are thus limited by the state space explosion problem. This work uses symbolic techniques in order to avoid this problem. It is based on Input Output Symbolic Transition Systems (IOSTS), which extend Input Output Labelled Transition Systems (IOLTS) by the use of variables and parameters. System specifications as well as test purposes, which are used in conformance testing to specify what aspects of the system have to be tested, are defined as IOSTS. In this way, it is possible to generate test cases without enumerating the specification's state space. The resulting test cases are symbolic and can be made executable by instantiation of their variables.

At first, an existing test case generation approach based on IOSTS is studied. It uses reachability analyses and is implemented in the tool STG (Symbolic Test Generator). However, this approach shows weaknesses for some kinds of systems. Thus, this work presents an alternative way of generating test cases from IOSTS. Since only a part of the existing STG tool set causes problems, the new approach does not start from scratch, but reuses basic functionality concerning IOSTS from STG. Above all, the new approach replaces STG's problematic parts concerning the test case selection, which is based on reachability analyses. Instead, it employs symbolic execution of IOSTS, which requires SAT solving. Furthermore, an algorithm to select test cases from the resulting symbolic execution trees is presented. Finally, a prototype implementation of the new approach is discussed and evaluated in three case studies.

Keywords: Model-Based Testing, Conformance Testing, Automated Test Case Generation, Input Output Symbolic Transition Systems, Symbolic Execution, SMT Solving

Kurzfassung

Testen ist eine entscheidende sowie arbeitsintensive Aufgabe in der Softwareentwicklung. Um die Effizienz zu erhöhen, die Kosten zu senken, und Testen damit praktikabler zu machen, wird eine weitgehende Automatisierung angestrebt. Generell können alle Phasen des Testens zumindest teilweise automatisiert werden. Diese Arbeit befasst sich mit der Automatisierung der Testfallgenerierungsphase.

Heutzutage gibt es bereits einige Tools zur Automatisierung der modellbasierten Testfallgenerierung. Da die meisten von ihnen den Zustandsraum der Spezifikation explizit berechnen, gelangen sie sehr schnell an die Grenzen der Speicherkapazität (State Space Explosion Problem). Diese Arbeit wendet symbolische Verfahren an um dieses Problem zu vermeiden. Dabei werden symbolische Transitionssysteme mit Input/Output (engl. Input Output Symbolic Transition Systems, IOSTS) verwendet. Sie stellen eine Erweiterung von IOLTS (Input Output Labelled Transition Systems) mittels Variablen und Parametern dar. Sowohl System-Spezifikationen als auch *Test Purposes*, die beim Konformitätstesten angeben welche Teile des Systems getestet werden, werden als IOSTS modelliert. Auf diese Weise können Testfälle generiert werden, ohne den Zustandsraum der Spezifikation zu enumerieren. Die resultierenden Testfälle sind symbolisch und können in ausführbare Testfälle umgewandelt werden, indem ihre Variablen instantiiert werden.

Zunächst wird ein bereits bestehender Ansatz zur Testfallgenerierung basierend auf IOSTS untersucht. Dieser Ansatz verwendet Erreichbarkeitsanalysen und ist in dem Tool STG (Symbolic Test Generator) implementiert. Allerdings zeigt dieser Ansatz Schwächen bei einigen Arten von Systemen. Deshalb stellt diese Arbeit eine alternative Möglichkeit zur Testfallgenerierung basierend auf IOSTS vor. Da nur bestimmte Teile von STG Probleme aufweisen, werden im neuen Ansatz Grundfunktionen bezüglich IOSTS wiederverwendet, die bereits in STG implementiert sind. Nur die problematischen Teile, die vor allem die Testfallauswahl mittels Erreichbarkeitsanalysen betreffen, werden ersetzt. Stattdessen wird ein Verfahren zur symbolischen Ausführung von IOSTS eingesetzt, das SAT-Solving erfordert. Somit wird ein Baum (engl. Symbolic Execution Tree) berechnet, aus dem ein Testfall ausgewählt wird. Der Algorithmus zur Testfallauswahl wird in dieser Arbeit ebenso präsentiert wie ein Prototyp, der den neuen Testfallgenerierungsansatz implementiert. Zum Abschluss wird der neue Ansatz mittels dreier Fallstudien evaluiert.

Schlagworte: Modellbasiertes Testen, Konformitätstesten, Automatisierte Testfallgenerierung, Symbolische Transitionssysteme mit Input/Output (engl. Input Output Symbolic Transition Systems), Symbolische Ausführung, SMT-Solving

Statutory Declaration

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

....	....
place, date	(signature)

Eidesstattliche Erklärung

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

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

Acknowledgements

I would like to thank numerous people who supported me while I was writing this thesis and accompanied me throughout my preceding studies.

I am grateful to Prof. Franz Wotawa, head of the Institute for Software Technology, who supported this work.

I also want to express my gratitude to my advisor, Dr. Bernhard Aichernig. Long before starting this work, he has sparked my interest in software testing and allowed me to work in this area. During the creation of this work, he always had an open ear for my questions and problems and helped me with his constructive ideas.

Furthermore, I want to thank Martin Weiglhofer, who has also made a valuable contribution to this work. Through his expertise on the formal specification of protocols, he made the case studies of this work possible. Moreover, he had always time to discuss implementation issues.

I would like to address special thanks to my boyfriend Robert Könighofer. Throughout all the ups and downs, which were connected with the writing of this work, he has always motivated and believed in me. Last but not least, I want to thank my parents Rupert and Helene Jöbstl. They have supported my decision to study and have encouraged me all the time.

Elisabeth Jöbstl
Graz, Austria, September 2009

Danksagung

Ich möchte mich herzlich bei allen bedanken, die mich bei dieser Arbeit sowie im vorausgehenden Studium unterstützt und begleitet haben.

Beim Vorstand des Instituts für Softwaretechnologie, Prof. Franz Wotawa, möchte ich mich für die Unterstützung meiner Arbeit bedanken.

Insbesondere gilt mein Dank auch meinem Betreuer, Dr. Bernhard Aichernig. Lange bevor ich mit dieser Arbeit begonnen habe, hat er mein Interesse für das Software-Testen geweckt und mir ermöglicht mich auf diesem Gebiet zu betätigen. Während der Erstellung dieser Arbeit hatte er immer ein offenes Ohr für meine Fragen und Probleme und half mir immer wieder mit konstruktiven Denkanstößen weiter.

Auch bei Martin Weiglhofer, der ebenfalls einen wertvollen Beitrag zu dieser Arbeit geleistet hat, möchte ich mich herzlich bedanken. Durch sein Expertenwissen über das formale Spezifizieren von Protokollen wurde die Fallstudie dieser Arbeit ermöglicht. Weiters hatte er immer Zeit um mit mir Detailfragen zur Implementierung zu diskutieren.

Ganz besonderer Dank gilt meinem Freund Robert Könighofer. Während aller Höhen und Tiefen, die mit dem Schreiben dieser Arbeit verbunden waren, hat er mich immer wieder motiviert und mir gezeigt, dass er an mich glaubt. Nicht zuletzt möchte ich meinen Eltern Rupert und Helene Jöbstl danken, die mir durch ihre fortwährende moralische und finanzielle Unterstützung das Studium und diese Arbeit ermöglichten.

Elisabeth Jöbstl
Graz, Österreich, im September 2009

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

"Testing is the process of executing a program with the intent of finding errors."

This definition comes from Glenford Myers [58], who also stated that complete testing is not possible, i.e., finding all errors of a program is infeasible. Hence, testing is a complex and challenging task. Already in the 1970s, it was known that about 50 % of the elapsed time and more than 50 % of the total costs of a software project are spent in testing [58]. It is also known that the later a software error is detected, the higher are the costs for fixing it [43]. Testing is not only expensive - the costs for testing are also often underestimated and poorly planned [18], although undetected software errors potentially cause enormous costs or even endanger human lives.

In 1996, the Ariane 5 rocket got uncontrollable and had to be exploded just 40 seconds after initiation of the flight sequence. The software system calibrated for the Ariane 4 rocket has been reused without proper testing. This error caused costs in the amount of $ 370 million [32]. Until 2005, a therapy planning software at the National Cancer Institute of Panama City caused at least 18 deaths. Although incorrect data sequences have been input to the software, it did not alert the user and calculated improper dosages of radiation for cancer patients [15].

Automation can be applied in order to improve the efficiency of testing, to lower its costs, and hence to make it more viable. In general, all phases of testing, i.e., test design, test execution, and test evaluation, can be supported or even fully automated by testing tools. This work focuses on the automation of test case generation via model-based testing (see Section 1.2).

Furthermore, we generate *symbolic* test cases. In this way, the state space explosion problem, with which many other test case generation tools are struggling, is prevented (see Section 1.3.2). Symbolic test cases are also easier to understand for humans, since they are less complex than concrete tests.

The relatively low number of industrial case studies indicates that model-based testing is not very popular in industrial software projects yet [1]. This thesis aims at showing that model-based testing can be successfully applied to industrial-sized applications. In the future, the test case generation tool of this work shall be combined with a software for translating UML State Charts into IOSTS, which serve as models for our approach [72]. By allowing the user to define system specifications and test purposes via UML State Charts, an additional support for modelling large systems is given. Furthermore, a tool for executing the generated test cases is under development while this thesis is written. Hence, a framework for covering the model-based testing process including model creation, test case generation, and test case execution shall be provided.

1.2 Model-Based Testing

There are various definitions of model-based testing (MBT) in the literature. All of them involve the use of a model for testing, whereas the modelled subject and the aim of MBT can differ. In the following, a few perceptions of MBT will be presented. Of course, this list cannot be exhaustive.

Frantzen et al. [36] describe MBT as a black-box testing technique. The goal of MBT is to test whether a system under test (SUT) conforms to a formal specification (model) of the SUT. The model can be used for automatic test case generation and as an oracle, i.e., test result evaluation can be performed automatically, which requires a formal conformance relation.

Utting and Legeard [77] relate MBT very closely to *automation*. They focus on MBT in terms of generating test cases with oracles based on behavioural models. MBT covers the generation of (a) input values, (b) call sequences and (c) oracles for checking the test results, whereas the automating aspect is emphasized. The authors define MBT as *"the automation of the design of black-box tests"*.

Similarly, Pretschner and Philipps [64] describe the main idea of model-based testing as the use of models to express the intended behaviour of a system. These models are then used to derive test cases including input and expected output, which can be run on the SUT. Unlike the two already introduced approaches [36; 77], the authors do not identify automation as a characteristic of MBT. The manual derivation of test cases from a model also belongs to MBT.

Binder [13] claims that testing should always be model-based. Testing can be seen as searching for bugs. Exhaustive testing is infeasible. Hence, testing has to be systematic, focused, and automated. According to Binder [13], MBT has all of these three attributes.

Three further approaches, which are sometimes interpreted as MBT, are mentioned by Utting and Legeard [77]:

- The generation of test input data from a domain model.
- The generation of test cases from a model of the environment of the SUT.
- The generation of test scripts from abstract tests.

This work does not correspond to these three approaches, but deals with MBT as defined above: the generation of test cases with oracles based on behavioural models.

1.2.1 The Model-Based Testing Process

According to Utting and Legeard [77], the process used in MBT consists of the following five steps:

1. Model creation of the SUT and/or its environment.
2. Generation of abstract test cases from the model.
3. Generation of concrete, executable test cases by concretion of the abstract test cases.
4. Execution of the concrete test cases on the SUT and assignment of verdicts.
5. Analysis of the test results.

Figure 1.1 was influenced by Figure 2.4 of Utting and Legeard [77] and Figure 10.1 of Pretschner and Philipps [64]. It depicts the MBT process as defined above and completes it by inserting model validation, which should accompany the creation of the model. Additionally, a test case specification was introduced, which serves as selection criterion [64].
Hence, the process of model-based testing is the following [77]:

1. **Model Creation and Validation**
   Based on the requirements, the system and/or its environment has to be modelled. According to Stachowiak [71], a model has the following three properties:

   - A model is a *mapping* from a concrete ("original") into a more abstract ("model") world.
   - A model serves a specific *purpose*.
   - A model is a *simplification*. It does not reflect all attributes of the concrete world.

   Hence, the formal model created in this step should be much simpler and smaller than the SUT. It should be focused on the aspects of the SUT which shall be tested, and details should be abstracted. Since the model will be used for verification of the SUT, it has to be validated. It is necessary to check whether the model correctly represents the user requirements.

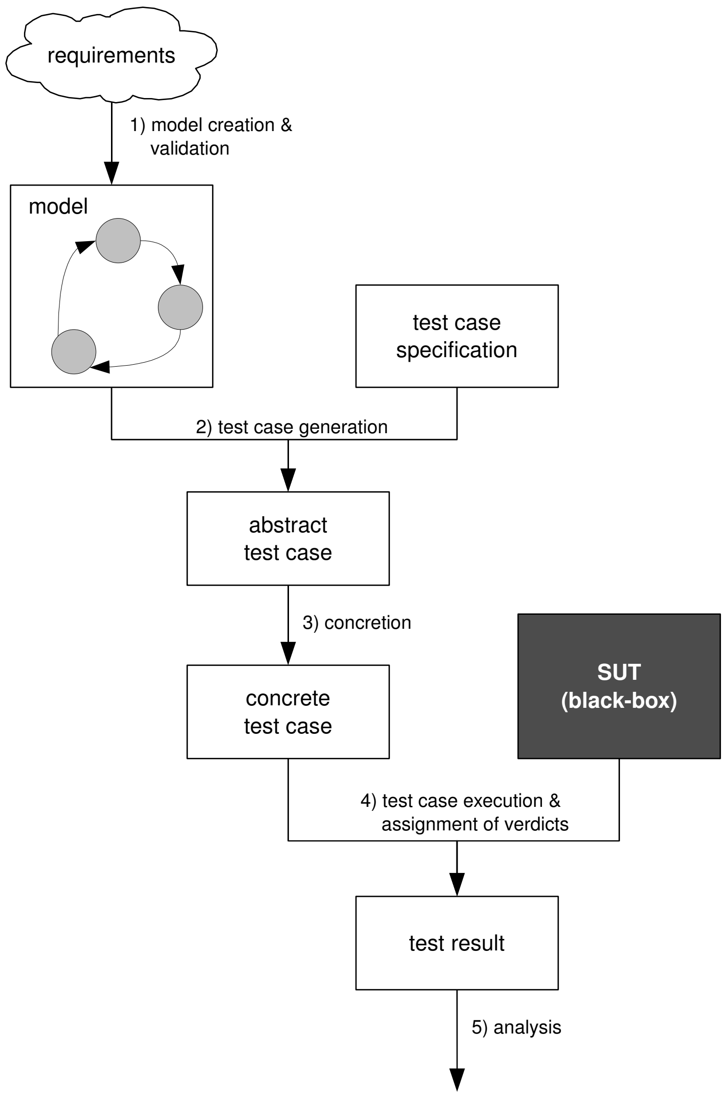


Figure 1.1: The model-based testing process: Based on the requirements, a formal model is created and validated. The model and a test case specification are used for the automatic generation of abstract test cases. After concretion of the abstract test cases, the tests are executed on the SUT and verdicts are assigned. Finally, the test results are analyzed.

2. **Test Case Generation**
   The created model is now used to automatically generate test cases, which are sequences of operations of the model. To define which tests shall be generated from the model, a test case specification is necessary. Without such a specification, usually an unlimited number of test cases could be generated. For automatic test case generation, different algorithms and heuristics are used. Since the model is a simplification of the SUT, the resulting test cases are not detailed enough to be directly executable. Hence, they are called *abstract test cases*.

3. **Concretion**
   To make the abstract test cases executable, they have to be concretized. The goal of this step is to close the gap between the abstract test cases and the concrete SUT. Figure 1.2 is based on a figure by Pretschner and Philipps [64] and illustrates how concretion and abstraction are used to connect the "model" world with the "original" world. The abstract input i, which is defined for the "model" world, has to be transferred into the "original" world. In order to be a valid input for the SUT, the abstract input i has to be concretized via the concretion function γ.

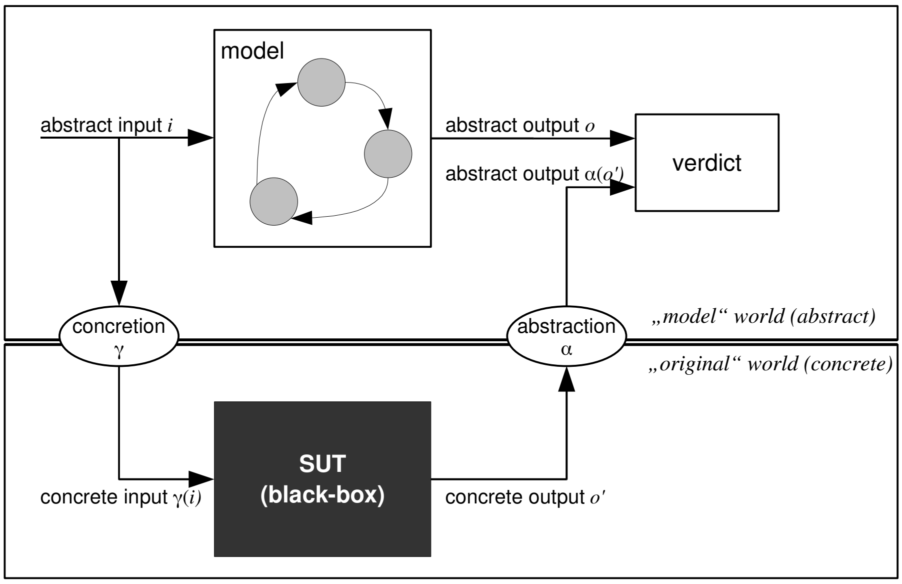


Figure 1.2: Abstraction and concretion are needed in model-based testing to connect the “model” world with the “original” world.

4. **Test Case Execution and Assignment of Verdicts**
By now, the concrete test cases can be executed. Again, Figure 1.2 will be used for illustration. The SUT processes the concrete input $\gamma(i)$ and produces some concrete output o'. When executing a test case, the actual output of an SUT has to be compared to the expected output to assign verdicts for each test. In model-based testing, the expected output o can be derived from the model, which means that it is defined in the “model” world. Hence, the concrete output from the SUT has to be abstracted via the abstraction function α in order to be compared to the abstract expected output o to generate a verdict.

5. **Analysis of the Test Results**
Finally, the results of the test executions have to be analyzed. For each test reporting a failure, the fault causing this failure has to be found. This fault is not necessarily located in the SUT. It could lie within the implementation of the concretion function γ. It could also be in the test case, which means that it would be in the model used for test case generation. Thus, the analysis of the test results also helps validating the model [7; 77].

This work focuses on the generation of test cases (Step 2 of the MBT process). The models used as input for Step 2 have been derived from already existing formal specifications defined in other modelling languages than the one used in this thesis. The further steps, i.e., concretion, test case execution and the assignment of verdicts, as well as the analysis of the test results, are out of our scope.

1.2.2 Benefits and Limitations of Model-Based Testing

According to Utting and Legeard [77], MBT achieves good results in fault detection in the SUT. In several case studies, the model-based testing approach found the same number of errors in the SUT or even more compared to manually designed tests. The authors also state that the quality of model-based test cases is better than the quality of manually designed tests. Manual testing requires an experienced tester, who is good at guessing sources of error. In addition, the manual test design process is often not reproducible. Since MBT uses algorithms and heuristics to automatically generate test cases, the test design is more systematic and reproducible. Due to automation, it is possible to produce a great number of test cases, which can help to find more bugs.

The requirements for the SUT are typically informal and formulated in natural language. Hence, they are possibly ambiguous, incomplete, or contradictory. When building a model from the requirements to describe the intended behaviour of the system, problems in the informal requirements can be revealed. Since the model is formal, which means it has precise semantics, encountered problems in the requirements have to be resolved. The clarification of requirements issues is crucial, because each resolved requirements problem means less errors during design and implementation. The earlier an error is found, the cheaper it is to fix. Frequently changing requirements necessitate the adaption of tests. The update of test suites written manually is time-consuming. With MBT, this task is easier. Only the model has to be updated and the tests can be generated anew [77].

According to Utting and Legeard [77], traceability, which is *"the ability to relate each test case to the model, to the test selection criteria, and even to the informal system requirements"*, is improved by MBT. For example, this can help to optimize test execution. In the case of changes to the model, only tests affected by changes need to be executed again.

Since models are a simplified version of the SUT, they are easier to understand, validate, and maintain than the SUT [7]. Particularly, models make it easier to automatically generate test cases. According to many scientists, e.g., Hierons et al. [44] or Tretmans [76], this is actually the main benefit of explicit model building and model-based testing.

All of the above described advantages of MBT help to reduce testing costs and time. Nevertheless, MBT has also drawbacks, which may outweigh the benefits and may prevent the cost and time balance to be positive. MBT involves an extra effort compared to conventional software testing: model building, validation, and maintenance. The analysis of the failed tests is also more complex, because there exist several sources of error: the SUT, the model, and the implementation of the concretion function [64; 77].

Although good results have been achieved with MBT, it is not the silver bullet to find software bugs. The abstraction and modelling skills of the tester as well as the selection of the test case specification have a great impact on the success of MBT. This may cause additional training costs when MBT is deployed the first time [77].

According to Utting and Legeard [77], the main field of application of MBT in software testing is functional testing. Only little experience is available with MBT for other kinds of testing. MBT is not only limited to functional testing, but also by the type of SUT. Not in every case, MBT is applicable. Sometimes, the deployment of manual testing is easier and leads to better results. Experience is required to decide whether MBT or conventional testing is more beneficial for testing a certain SUT. In order to apply MBT to generate appropriate test cases, the requirements and models have to be updated continuously. Otherwise, wrong properties of the SUT will be tested.

This thesis will apply MBT to protocol implementations for which it is known to achieve good results. Nevertheless, it will also expose that not all kinds of model-based testing tools are successfully applicable to all classes of problems in protocol conformance testing.

1.3 Reactive Systems and Symbolic Testing

1.3.1 Reactive Systems

This work focuses on testing of reactive systems. Manna and Pnueli [57] define reactive systems as systems, which are repeatedly prompted by the environment and respond continuously to external inputs. Hence, they are maintaining an ongoing interaction with their environment instead of calculating some final result. According to Dams et al. [30], reactive systems involve concurrency and non-determinism in most cases. Examples of reactive systems are flight reservation systems, embedded systems, operating systems, communication protocols, smart cards, etc. [30; 80].

1.3.2 Symbolic Testing

According to Clarke et al. [25] and Rusu et al. [67], most existing tools for the automation of test generation (e.g., TGV [48]) do not explicitly consider program *data*. Hence, the enumeration of the specification's state space (see Pezzè and Young [60]) is necessary. This leads to several problems: Since all variables of the specification are instantiated with all of their possible values, the state space explosion problem (see Pezzè and Young [60]) is likely to emerge. The effect is a limitation in the usability of these tools [36]. Frantzen et al. [36] identify another disadvantage of the enumeration of the state space. It causes a loss of structure information and knowledge about data definitions and constraints, although this information could be used to successfully improve the test selection procedure. Furthermore, the generated tests are difficult to understand for humans [25]. Rusu et al. [67] argue that test cases, where all variables are instantiated do not conform to industrial practice, where test cases are programs with parameters and variables.

Symbolic techniques can help to avoid these problems. The model of the SUT, the test purpose (test case specification), and the resulting test are symbolic, i.e., they contain variables, which are not instantiated. This makes them easier to understand for humans. An enumeration of the state space is not necessary when applying symbolic techniques, which is a great benefit. Symbolic tests can be made executable simply by instantiating its variables [25].

Several tools, e.g., TGV [48] or TorX [9], use the input output conformance relation (*ioco*). The *ioco* relation is based on LTS (Labelled Transition Systems [51]) and hence not on a symbolic level. Frantzen et al. [36] lift the *ioco* relation onto a symbolic level. The resulting conformance relation is called *sioco* (symbolic input output conformance) and is based on STS (Symbolic Transition Systems). STS explicitly integrate data and data-dependent control-flow, which means that they provide variables and the ability to guard transitions. The goal of Frantzen et al. [36] is to construct a complete formal framework for symbolic testing.

Despite of all of the above mentioned advantages, it has to be kept in mind that problems can also occur with symbolic testing techniques. Although they use more compact representations, they may suffer from problems in single cases as will be shown in the course of this work.

1.4 Problem Statement and Outline of its Solution

The fact that only a relatively low number of industrial case studies exist indicates that model-based testing suffers from a lack of industrial acceptance [1]. In the case of the symbolic test generation tool STG, this is not astonishing, since merely small examples are presented in connection with STG.

The very first goal of this master's thesis was to specify the Session Initiation Protocol (SIP) Registrar (see Section 6.2) with Input Output Symbolic Transition Systems (IOSTS) and to use STG in order to generate test cases for it. Soon, we realized that STG was not capable of handling this particular large-sized specification. We suppose that STG's troubles with the SIP Registrar specification are caused by the use of BDDs, which may lead to problems with specifications that comprise large Boolean formulae (see Section 3.4).

In a next step, we thought about a solution to this problem and how to reuse the already prepared specification of the SIP Registrar. Hence, those parts of STG's approach which are working properly will be involved in our new test case generation methodology. The problematic part, which comprises the test case selection strategy of STG, will be substituted by an alternative technique. Symbolic execution of IOSTS and an algorithm for selecting test cases from the resulting symbolic execution trees replace STG's test case selection, which employs symbolic reachability and coreachability analyses. Chapter 4 gives a detailed description of our test case generation approach.

1.5 Structure of this Thesis

The rest of this master's thesis is structured in the following way. Chapter 2 deals with conformance testing. It will give a general overview of the topic as well as a deeper insight into conformance testing with IOSTS, a model for the specification of reactive systems that will be defined in the course of the chapter. Furthermore, the Triangle Type Checker example, which will serve as a running example throughout this thesis, will be introduced.

Chapter 3 is dedicated to STG, a tool for symbolic test case generation. Its test case generation algorithm as well as its test case execution process will be presented. Moreover, its usage will be explained and problems encountered during experimentation with STG will be discussed.

Chapter 4 presents our alternative approach for test case generation. Its relation to STG will be explained and its main aspects, which are the symbolic execution of IOSTS and an algorithm for test case selection from symbolic execution trees, will be elaborated. Finally, the benefits and limitations of our new approach will be discussed.

Chapter 5 addresses the implementation of our approach. The architecture of the developed prototype will be presented and its usage will be explained.

Chapter 6 will report about the results of our approach applied to three examples: the Triangle Type Checker, the Session Initiation Protocol (SIP) Registrar, and the Conference Protocol. The results of the three case studies will be summarized and a conclusion will be given.

Chapter 7 discusses related work. A relation to publications in the field of transition systems as well as symbolic execution will be established.

Chapter 8 will summarize and discuss the previous chapters. Finally, some proposals about future work will be addressed.

2 Conformance Testing with Symbolic Transition Systems

This chapter is intended to give an overview of conformance testing in general and in the context of symbolic transition systems. The conformance testing formalization introduced by Tretmans, which is based on LTS (Labelled Transition Systems), will be presented. Furthermore, the IOSTS (Input Output Symbolic Transition System) model, which will be used throughout this thesis, will be introduced and a conformance testing theory for IOSTS will be presented.

2.1 Conformance Testing

When relying on the specification of some product, it has to be assured that the product behaves exactly like defined in its specification. Consider the following examples:

- *Communication protocols:* When different implementations of the same communication protocol interact, interoperability is only granted if all implementations conform to the protocol specification.
- *Compilers:* Compilers have to correctly implement language standards. Otherwise the compiled program could fail because of assumptions the software developer has made regarding program compilation.

To find out whether a product is indeed the one that was defined in the specification, conformance testing is employed. In conformance testing, the specification is the starting point for test case design. Hence, a requirement for conformance testing is the existence of a complete, unambiguous, and consistent specification of the SUT [74].

Conformance testing checks whether the SUT conforms to its specification. Conformance can be defined as a relation between the observable behaviour of the SUT and the behaviour of the corresponding model, which serves as system specification. An SUT conforms to its specification if both, the SUT and the specification, show the same behaviour [63].

Conformance testing belongs to the category of functional testing approaches. An SUT is solely tested according to its specification. The internal structure of the SUT (the code) is not known. Hence, in conformance testing, the SUT is a black-box. Only the observable behaviour of the SUT, the interactions of the SUT with its environment, is testable. For testing the SUT, the tester has to play the role of the environment and communicate with the SUT [48; 74].

To standardize conformance testing, the International Organization for Standardization (ISO)[1] published the international standard ISO/IEC 9646: *Information technology - Open Systems Interconnection - Conformance testing methodology and framework* [47]. Originally, the standard was intended for testing communication protocols, but it turned out that it can also be applied for testing the conformance of other reactive systems [80].

Tretmans [74] formalized the first version of the ISO/IEC 9646 standard [47]. In 1991, the standard consisted of five parts:

- *Part 1: General concepts* - gives an introduction and specifies a general methodology for conformance testing.
- *Part 2: Abstract Test Suite specification* - introduces the process of generating system-independent conformance test suites.

[1] `http://www.iso.org` (last visit 2009-09-27)

- *Part 3: The Tree and Tabular Combined Notation (TTCN)* - defines the test notation used in the standard.
- *Part 4: Test realization* - covers test execution.
- *Part 5: Requirements on test laboratories and clients for the conformance assessment process* - specifies requirements on the test laboratory and the client. Additionally, the conformance assessment process is discussed.

According to the International Organization for Standardization, Part 1, 2, 4, and 5 have been revised in 1994. Part 3 has been revised in 1998. Furthermore, two more parts have been added to the standard:

- *Part 6: Protocol profile test specification* - has been published in 1994.
- *Part 7: Implementation Conformance Statements* - has been added in 1995.

2.1.1 The Conformance Testing Process

According to the first part of the ISO/IEC 9646 standard [47], the process of conformance testing consists of three main phases: test generation, test implementation, and test execution. In the following, each phase will be described in more detail according to Tretmans [74]. Figure 2.1 serves as illustration, where the implementation process of the SUT is depicted as a parallel task flow to test generation and test implementation.

1. **Test Generation**
   Test generation covers the systematic design of an abstract test suite for the SUT. The resulting test suite has the following properties: It is independent of any implementation and it has to be expressed in a well-defined test notation language. The test should be suitable for standardization and test all important details of the specification.

   Static conformance requirements express restrictions on the selection of optional parts of the specification. Test generation is based on the *dynamic* conformance requirements of the specification, which are requirements on the observable behaviour of the SUT. The test generation phase consists of three steps:

   (a) For each dynamic conformance requirement, one or more *test purposes* are defined. A test purpose describes what will be tested. It is used to decide about the satisfaction of a conformance requirement.
   (b) For each test purpose, a generic test case is derived. A generic test case expresses the corresponding test purpose in terms of operations on a high level.
   (c) For each generic test case, an abstract test case is created. In this step, restrictions due to the testing environment are considered and a test method is chosen. PCOs (Points of Control and Observation) are interfaces for the tester to control and observe the SUT. A test method specifies at which PCOs the tester can access the SUT. Four test methods have been standardized in part 2 of ISO/IEC 9646 [47].

   The test notation proposed by ISO/IEC 9646 is the semi-formal language TTCN (Tree and Tabular Combined Notation). In TTCN, input and output events occurring at the PCOs represent the behaviour of test cases. Each sequence is closed by a verdict.

2. **Test Implementation**
   Test implementation deals with the transformation of the abstract test suite of Phase 1 into an executable test suite. The executable test suite is tailored to a specific testing environment and a specific SUT. The test implementation phase consists of two steps:

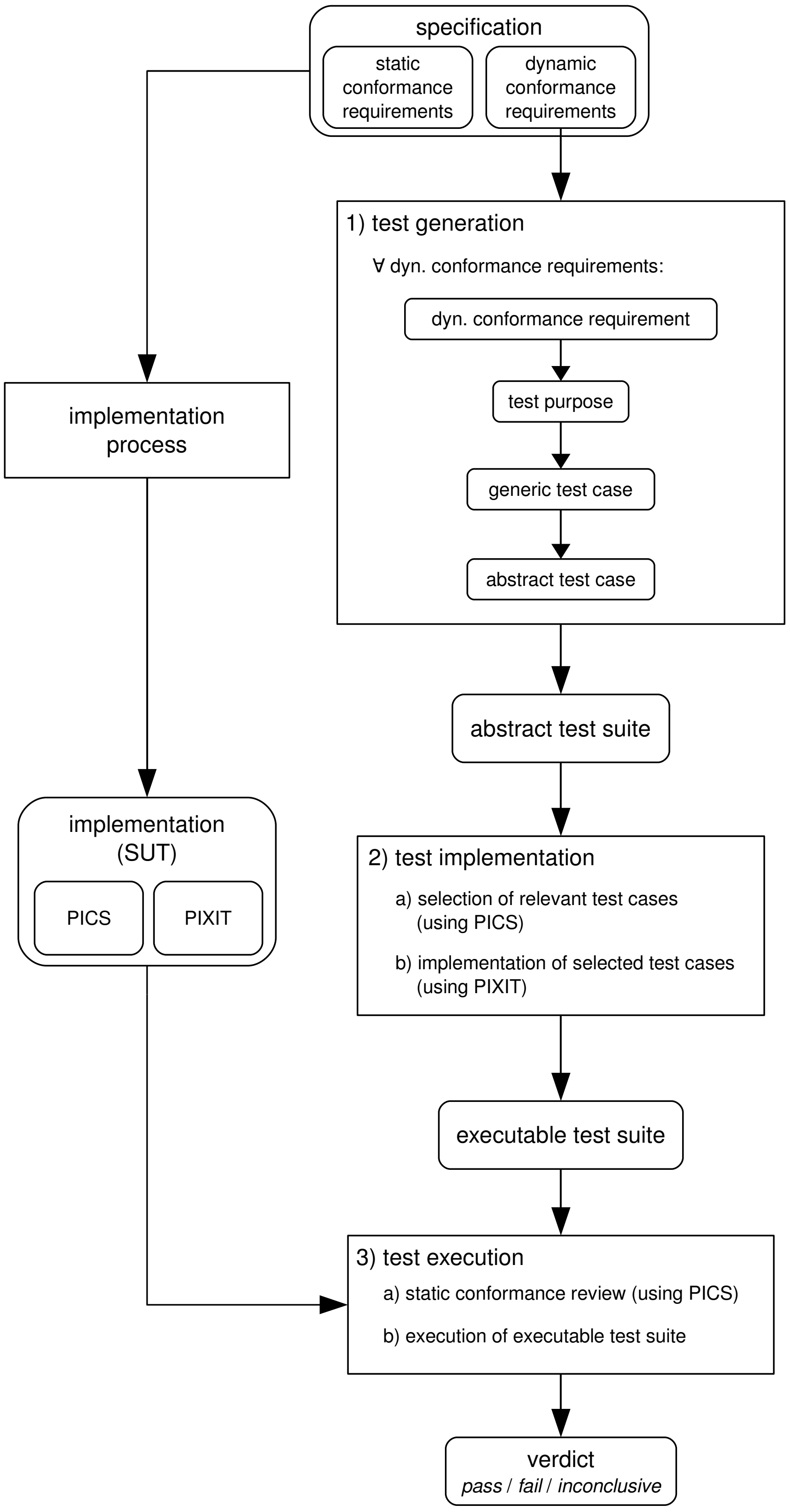

specification
static conformance requirements
dynamic conformance requirements
1) test generation
∀ dyn. conformance requirements:
dyn. conformance requirement
test purpose
generic test case
abstract test case
implementation process
abstract test suite
implementation (SUT)
PICS
PIXIT
2) test implementation
a) selection of relevant test cases (using PICS)
b) implementation of selected test cases (using PIXIT)
executable test suite
3) test execution
a) static conformance review (using PICS)
b) execution of executable test suite
verdict
pass / fail / inconclusive

Figure 2.1: The conformance testing process consists of three main steps: test generation, test implementation, and test execution. The implementation of the SUT can be seen as a task executed in parallel to test generation and test implementation.

   (a) Before starting with the implementation of the tests, a selection has to be made. Since the abstract test suite contains tests for the whole specification including optional parts, only tests which affect actually implemented features must be filtered. The implemented options are listed in the PICS (Protocol Implementation Conformance Statement) document provided by the supplier of the SUT.
   (b) For the implementation of the selected test cases, the PIXIT (Protocol Implementation eXtra Information for Testing) document, which is like the PICS document provided by the supplier of the SUT, is required. It contains information relevant for testing the SUT like parameter and timer values.

3. **Test Execution**
   Test execution means the execution of the executable test suite on some SUT to obtain a verdict about the conformance of the SUT to its specification. Two tasks have to be considered:

   (a) In a static conformance review, the PICS of the SUT is inspected for conformance with the static conformance requirements defined in the specification.
   (b) The executable test suite obtained in Phase 2 is executed on the SUT. The SUT's behaviour is observed and compared with the expected behaviour defined in each test case. This leads to a verdict, which can be *pass*, *fail*, or *inconclusive*. *Pass* means that the test purpose was satisfied and the SUT behaved correctly. *Fail* stands for non-conformance of the SUT for that specific test purpose. *Inconclusive* indicates that the SUT behaved correctly, but the test purpose could not be satisfied.

2.1.2 Conformance Testing Formalization

Tretmans [74] formalized the main concepts introduced in the ISO/IEC 9646 standard [47]. In the following, the most important aspects of his formalization will be presented.

In conformance testing, product specifications are the starting point for test case generation. Specifications can be either directly expressed as a set of requirements by means of logical languages or by using behavioural languages, which describe the observable behaviour of a system. The latter approach is more straightforward for humans and widely-used. Examples for logical languages are Z [70] or Temporal Logic [61]. Examples for behavioural specification languages are SDL [8], LOTOS [14], or Estelle [20].

In the following, $\mathcal{L}_R$ denotes the formal language in which requirements are expressed. $\mathcal{L}_{FDT}$ denotes the used formal description technique (FDT), which is behavioural in most cases. Conformance will be defined for both approaches, the logical and the behavioural approach.

Logical Approach When using the logical approach, a specification $Spec$ can be seen as a set of conformance requirements $r_i \in \mathcal{L}_R$, which forms a logical theory:

$$Spec = \{r_1, r_2, r_3, ...\}$$

The satisfaction of a requirement r by an implementation Imp is expressed by:

$$Imp \ \mathbf{sat} \ r$$

The set of requirements satisfied by an implementation Imp is denoted by $sats(Imp)$, which is a theory just like $Spec$:

$$sats(Imp) \quad =_{def} \quad \{r \in \mathcal{L}_R \mid Imp \ \mathbf{sat} \ r\}$$

In order to be a conforming implementation, the specification $Spec$ has to be a subtheory of $sats(Imp)$:

$$Spec \subseteq sats(Imp)$$

If an implementation Imp satisfies all requirements of the specification $Spec$, it is a conforming implementation:

$$Imp\ \textbf{sat}\ Spec \quad =_{def} \quad \forall r \in Spec : Imp\ \textbf{sat}\ r$$

According to ISO/IEC 9646 [47], an implementation which conforms to its specification must satisfy both static and dynamic conformance requirements. So, when the logical approach is used, the definition of conformance is straightforward. Conformance is defined as the satisfaction of the requirements defined by the specification.

Behavioural Approach A behavioural specification $Spec_B$ can be seen as a set of behavioural expressions $b_i \in \mathcal{L}_{FDT}$:

$$Spec_B = \{b_1, b_2, b_3, ...\}$$

In the course of using behavioural specifications, it is necessary to relate behaviour to requirements in order to define conformance. For this purpose, the relation **spec** is introduced. The fact that a behavioural expression b specifies a requirement r is denoted by:

$$b\ \textbf{spec}\ r$$

Analogously, the fact that a behavioural specification $Spec_B$ specifies a requirement r is written as:

$$Spec_B\ \textbf{spec}\ r$$

Now, conformance can be defined for a behavioural specification $Spec_B$:

$$Imp\ \textbf{conforms-to}\ Spec_B \quad =_{def} \quad \forall r \in \mathcal{L}_R : Spec_B\ \textbf{spec}\ r \rightarrow Imp\ \textbf{sat}\ r$$

According to this definition, conformance is a relation between implementations and behaviour specifications:

$$\textbf{conforms-to} \subseteq \textit{IMP} \times \mathcal{L}_{FDT}$$

IMP denotes the set of all possible implementations. If *IMP* and $\mathcal{L}_{FDT}$ are fixed, the definition of conformance is still variable since $\mathcal{L}_R$, **spec** and **sat** still have to be defined.

Test Hypothesis For test validation purposes, a formalism for modelling implementations $\mathcal{L}_{IMP}$ is introduced. It is assumed that each implementation Imp can be represented by a model $Imp_M \in \mathcal{L}_{IMP}$. This assumption is referred to as test hypothesis in literature [11]. It states that the implementation under test is close enough to its specification, so that testing becomes reasonable. A natural choice for $\mathcal{L}_{IMP}$ is $\mathcal{L}_{FDT}$.

Implementation Relation When using the behavioural approach, conformance could also be characterized solely on the basis of behavioural expressions, where requirements are not explicitly taken into account any more. $\mathcal{L}_R$, **spec** and **sat**, which were necessary for defining the conformance relation, become unessential for studying conformance. When conformance is based on the comparison of observable behaviours of the specification and the implementation, conformance is seen as a relation $\leq_{\mathcal{R}} \subseteq \mathcal{L}_{FDT} \times \mathcal{L}_{FDT}$ called *implementation relation*.

An implementation relation $\leq_{\mathcal{R}}$ is compatible with a requirement language $\mathcal{L}_R$ as well as the relations **spec** and **sat** if for all behavioural specifications $Spec_B$ and implementations Imp:

$$Imp \leq_{\mathcal{R}} Spec_B \quad \leftrightarrow \quad Imp\ \textbf{sat}\ specs(Spec_B),$$

whereas $specs(Spec_B)$ denotes the set of requirements specified by the behavioural specification $Spec_B$. Its definition is:

$$specs(Spec_B) \quad =_{def} \quad \{r \in \mathcal{L}_R \mid Spec_B\ \textbf{spec}\ r\}$$

Note that inconclusive does not mean that an implementation has shown incorrect behaviour. It indicates that the SUT behaved correctly, but the test purpose could not be satisfied.

Conformance Tests and Test Suites The set of test cases is modelled by a test notation $\mathcal{L}_T$. The application of a test case $tc \in \mathcal{L}_T$ to an implementation $Imp \in \mathcal{L}_{FDT}$ is expressed by the function:

$$\mathit{apply} : \mathcal{L}_T \times \mathcal{L}_{FDT} \rightarrow \{\textbf{pass}, \textbf{fail}, \textbf{inconclusive}\}$$

If an implementation Imp passes a test case tc, this is expressed by:

$$Imp\ \textbf{passes}\ tc \quad =_{def} \quad \mathit{apply}(tc, Imp) \neq \textbf{fail}$$

For whole test suites $TS \subseteq \mathcal{L}_T$, the relation **passes** is defined in the following way:

$$Imp\ \textbf{passes}\ TS \quad =_{def} \quad \forall tc \in TS : Imp\ \textbf{passes}\ tc$$

The relation **fails** is defined as the negation of **passes**:

$$Imp\ \textbf{fails}\ tc \quad =_{def} \quad \neg(Imp\ \textbf{passes}\ tc)$$

$$Imp\ \textbf{fails}\ TS \quad =_{def} \quad \neg(Imp\ \textbf{passes}\ TS)$$

Conformance test suites have the following properties:

- **Soundness:**
  A test suite TS is sound if only incorrect implementations are identified as non-conforming:

  $$\forall Imp \in \mathcal{L}_{FDT} : Imp \leq_{\mathcal{R}} Spec_B \rightarrow Imp\ \textbf{passes}\ TS$$

- **Exhaustiveness:**
  A test suite TS is exhaustive if it correctly identifies all non-conforming implementations:

  $$\forall Imp \in \mathcal{L}_{FDT} : \neg(Imp \leq_{\mathcal{R}} Spec_B) \rightarrow Imp\ \textbf{fails}\ TS$$

- **Completeness:**
  A test suite TS is complete if it rejects all and only non-conforming implementations, i.e., if it is sound and exhaustive.

2.2 IOSTS - Behavioural Models

IOSTS (Input Output Symbolic Transition System) belong to the category of behavioural specification techniques (see Section 2.1). According to Jeannet et al. [49], an IOSTS is a data structure allowing to model infinite-state transition systems. It is used in STG for the representation of system specifications, test purposes, and generated test cases. Furthermore, implementations can be modelled by IOSTS which are unknown except for their interface (black-box testing) [67].

The IOSTS model is an extension of the IOLTS (Input Output Labelled Transition System) model. IOLTS are rooted LTS (Labelled Transition Systems [51]) with distinguished inputs and outputs [49]. IOLTS were handled in several publications by different authors, e.g., by Aichernig et al. [2], Rusu et al. [68], and Tretmans [76]. The main difference between IOSTS and IOLTS is the fact that IOSTS are a representation on a symbolic level, whereas IOLTS deal with concrete data values. This symbolic representation is enabled by enriching IOSTS with variables and parameters and has the advantage of not having the state space explosion problem [67] (see also Section 1.3.2).

There are several sources in literature defining the IOSTS model, e.g., Jeannet et al. [49; 50] or Rusu et al. [67]. This section is supposed to give an overview of the most important aspects of IOSTS. In the following, a definition of Input Output Symbolic Transition Systems is given and the associated semantics will be explained with the aid of Labelled Transition Systems (LTS). Furthermore, different types of IOSTS will be distinguished and conformance testing (see Section 2.1) on IOSTS will be introduced. For illustration, a running example called "Triangle Type Checker" will be used throughout this section.

2.2.1 Definition of IOSTS

Rusu et al. [67] define an IOSTS $\mathcal{I}$ as a tuple $\langle D, \Theta, Q, q_0, \Sigma, \mathcal{T} \rangle$ with the following components:

- D is a set of typed data, which can be expressed by the formula $D = V \cup P \cup M$, whereas V is the set of variables, P is the set of system parameters, and M is the set of messages of $\mathcal{I}$. It is important not to mix up P and M. P represents the parameters of the whole specified system, hence, P denotes the *system parameters*. The messages M represent the arguments of the system's actions specified by Σ, which would be called parameters in other language notations. V, P, and M are pairwise disjoint. D is nonempty and finite.
- Θ is the initial condition on elements in $V \cup P$.
- Q is the nonempty and finite set of locations.
- $q_0 \in Q$ is the initial location.
- Σ is the nonempty, finite alphabet, which can be expressed by the formula $\Sigma = \Sigma^i \cup \Sigma^o \cup \Sigma^{int}$, whereas Σ^i is the set of input actions, Σ^o is the set of output actions, and Σ^{int} is the set of internal actions. Each action a is possibly carrying typed messages. Hence, it has a signature $sig(a) = \langle \vartheta_1, ...\vartheta_k \rangle$, which is a tuple of its message types. Signatures may be the empty tuple, which is particularly the case for internal actions.
- $\mathcal{T}$ is the set of transitions. A transition is a tuple $\langle q, a, \mu, G, A, q' \rangle$ consisting of:
  - $q \in Q$ is the location from which the transition starts.
  - $a \in \Sigma$ is the action of the transition.
  - μ is the tuple of messages for the action a of the transition. The message types have to be in line with the signature of a. Messages are only visible in the transition in which they are used. They behave like local variables [49].
  - G, called the guard of the transition, is a Boolean expression built upon the elements of $V \cup P \cup \mu$.
  - A is a set of expressions and represents the transition's assignments. It is assigning each variable of V a new value. This new value is an expression over the elements of $V \cup P \cup \mu$. Self-assignments of variables are made implicitly and do not have to be written explicitly in STG. For each variable in V, there is exactly one assignment.
  - $q' \in Q$ is the destination location of the transition.

2.2.2 Example: Triangle Type Checker

The following simple example is called "Triangle Type Checker". It will be continued throughout this thesis to illustrate theoretically explained concepts. Originally, the idea for this example comes from Myers [58]. The example IOSTS reads three input values of type integer, representing the three side lengths of a triangle. The IOSTS determines whether these three sides form a valid triangle. If they do, the type of the triangle (equilateral, isosceles, or scalene) is decided. If one of the side lengths is negative or zero, *NotPositive* is sent. If the three side lengths do not form a valid triangle, *NotTriangle* is reported.

Figure 2.2 shows an IOSTS as defined above: the specification of the Triangle Type Checker printed by STG. Locations of the IOSTS are depicted as ellipses. Arrows between these ellipses illustrate transitions. The guards, the actions with their messages, and the assignments of the transitions are depicted inside rectangles between these arrows. Input actions are followed by *?*, output actions by *!* in the graph of Figure 2.2 as well as in the STG syntax (see the STG Reference Manual[2]). For example, the transition from location *Readabc* to *CheckPositive* has no guard (*true*), waits for an input of three integers ($Read?(p, q, r)$), and assigns these three input values to the IOSTS variables a, b and c.

[2]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_4.html` (last visit 2009-09-27)

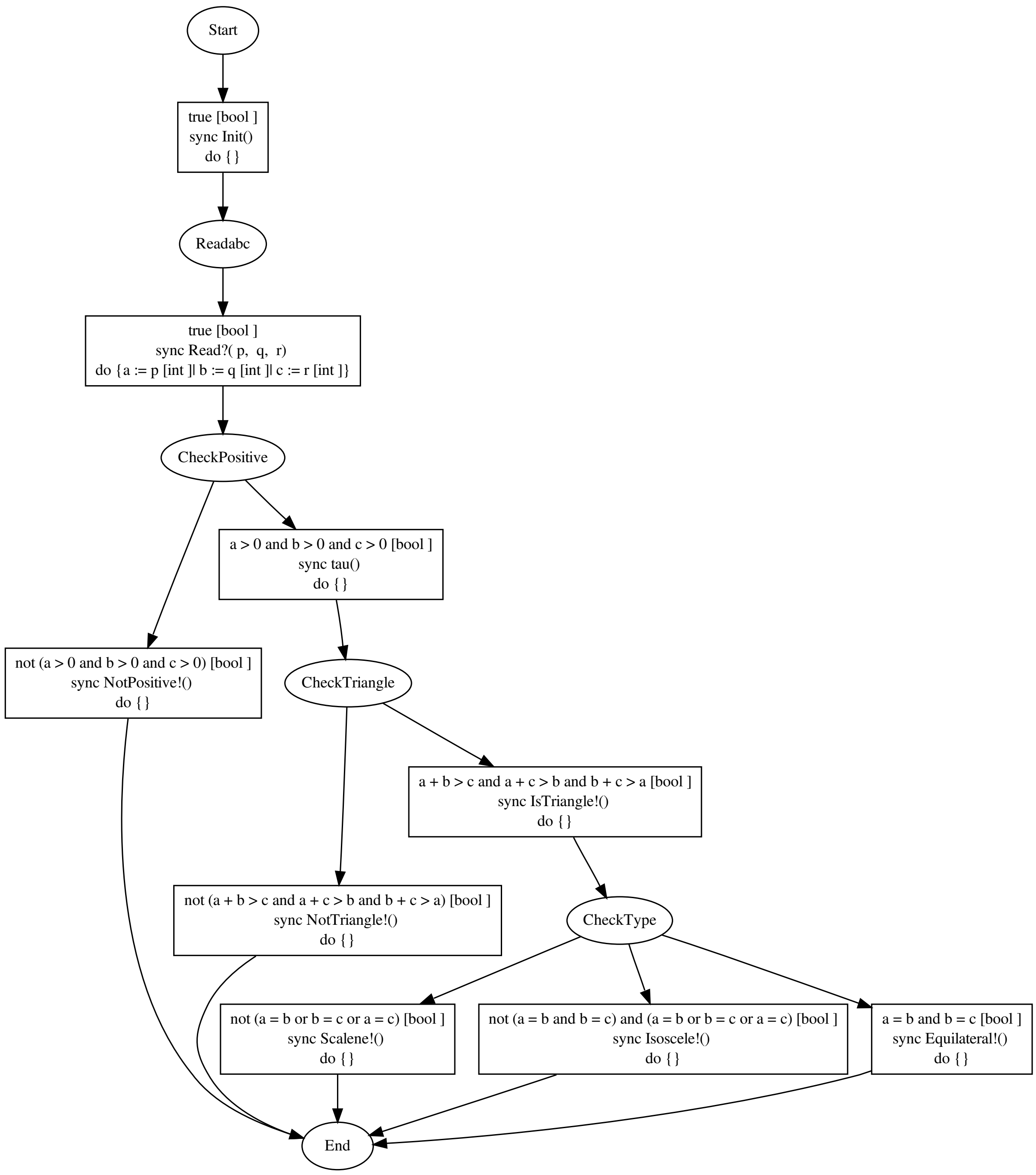

Start
true [bool]
sync Init()
do { }
Readabc
true [bool]
sync Read?(p, q, r)
do {a := p [int]| b := q [int]| c := r [int]}
CheckPositive
a > 0 and b > 0 and c > 0 [bool]
sync tau()
do { }
not (a > 0 and b > 0 and c > 0) [bool]
sync NotPositive!()
do { }
CheckTriangle
a + b > c and a + c > b and b + c > a [bool]
sync IsTriangle!()
do { }
not (a + b > c and a + c > b and b + c > a) [bool]
sync NotTriangle!()
do { }
CheckType
not (a = b or b = c or a = c) [bool]
sync Scalene!()
do { }
not (a = b and b = c) and (a = b or b = c or a = c) [bool]
sync Isoscele!()
do { }
a = b and b = c [bool]
sync Equilateral!()
do { }
End

Figure 2.2: This graph depicts the IOSTS specifying the Triangle Type Checker system. It waits for three integer values, which represent the side lengths of a triangle and determines whether the triangle is valid. If it is, the type of the triangle is decided. If one of the integers is negative or zero, "NotPositive" is sent.

Table 2.1 lists the components of an IOSTS as defined above for the IOSTS of Figure 2.2. The set of variables V of the IOSTS contains three elements. The set of IOSTS system parameters P is empty. The set of messages M contains three elements. Since the set of typed data D is the union of these three sets, D contains six elements. The initial condition of the IOSTS is *true*. The IOSTS consists of six locations, whereas *Start* is the initial location. One input action, six output actions, and one internal action are defined for the IOSTS. The transitions of the IOSTS are not listed in the table, because a textual description would be long and confusing compared to the visual representation.

The STG specification file for the above described IOSTS can be found at the STG web page[3].

1	V	$=$	$\{a, b, c\}$
2	P	$=$	$\emptyset$
3	M	$=$	$\{p, q, r\}$
4	D	$=$	$\{a, b, c, p, q, r\}$
5	Θ	$=$	$true$
6	Q	$=$	$\{Start, Readabc, CheckPositive, CheckTriangle, CheckType, End\}$
7	q_0	$=$	$Start$
8	Σ^i	$=$	$\{Read\}$
9	Σ^o	$=$	$\{NotPositive, NotTriangle, IsTriangle, Scalene, Isoscele, Equilateral\}$
10	Σ^{int}	$=$	$\{tau\}$

Table 2.1: A list of the main components of the IOSTS specifying the Triangle Type Checker, which is depicted in Figure 2.2.

2.2.3 Semantics of IOSTS

The semantics of IOSTS can be explained by means of LTS (Labelled Transition Systems [51]), since the IOSTS model extends LTS with variables and parameters [67]. Therefore, IOSTS deal with symbolic values, whereas LTS handle concrete values. The following terminology, which is necessary for describing the semantics of an IOSTS $\mathcal{I} = \langle D, \Theta, Q, q_0, \Sigma, \mathcal{T} \rangle$, was defined by Rusu et al. [67]:

Valuation $Val(D')$ denotes all (type-consistent) valuations of D', whereas D' is a subset of the IOSTS data D ($D' \subseteq D$). It associates to each element of D' a value of correct type.

State A state s is a pair consisting of a location and a valuation for the IOSTS variables and system parameters ($s = \langle q, v \rangle$, whereas $q \in Q$ and $v \in Val(V \cup P)$). The set of states of an IOSTS is denoted by S.

Initial State An initial state s_0 is a state fulfilling the following conditions: its location is the initial location and its valuation entails the initial condition ($s_0 = \langle q_0, v_0 \rangle$, whereas $v_0 \models \Theta$). The set of initial states is called S_0.

Valued Input/Output A valued input is a tuple consisting of an input action and a list of type-consistent values for the action's messages ($valued\ input = \langle a, v_1, ..., v_l \rangle$, whereas $a \in \Sigma^i$, $sig(a) = \langle \vartheta_1, ..., \vartheta_l \rangle$ and $type(v_i) = \vartheta_i$ for $1 \leq i \leq l$). Valued outputs are defined in the same way, except of having an output instead of an input action. Υ denotes the set of valued inputs, Ω the set of valued outputs.

Value of a Data Item Given a valuation $v \in Val(D)$ and an element $d \in D$, then $v(d)$ denotes the value for the data item d.
Assuming D' and D'' are disjoint subsets of data ($D', D'' \subseteq D$ and $D' \cap D'' = \emptyset$) and v and w are two of its valuations ($v \in Val(D')$, $w \in Val(D'')$), then the valuation $v \cdot w(D' \cup D'')$ is defined by

- $v \cdot w(d) = v(d)$, if $d \in D'$
- $v \cdot w(d) = w(d)$, if $d \in D''$.

[3]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_10.html` (last visit 2009-09-27)

Value of an Expression Given an expression e composed of elements in D and a valuation $v \in Val(D)$, then $v(e)$ is defined by the value resulting after the replacement of all elements $d \in D$ occurring in e by their values $v(d)$.

By now, the semantics of an IOSTS $\mathcal{I} = \langle D, \Theta, Q, q_0, \Sigma, \mathcal{T}\rangle$ can be defined by an LTS $\mathcal{L} = \langle S, S_0, L, \rightarrow\rangle$ consisting of the following components [67]:

- S denotes the set of states, S_0 the set of initial states as they were defined above.
- L, the set of labels of the LTS, is defined as $L = \Upsilon \cup \Omega \cup \Sigma^{int}$.
- The transition relation $\rightarrow$ is a set of tuples of the form $\langle s, l, s'\rangle$ whereas $s, s' \in S$ and $l \in L$.

Given the following information $\langle q, a, \mu, G, A, q'\rangle \in \mathcal{T}$

$v, v' \in Val(V \cup P)$

$w \in Val(\mu)$

$v \cdot w(G) = true$

$\forall x \in V : v'(x) = v \cdot w(A^x)$

$\forall x \in P : v'(x) = v(x)$

the LTS transition relation $\rightarrow$ is defined as $\langle q, v\rangle \xrightarrow{\langle a,w\rangle} \langle q', v'\rangle$.

Thus, there is a transition in the LTS $\mathcal{L}$ from state $\langle q, v\rangle$ to $\langle q', v'\rangle$ with action a and a valuation of the action's messages w if there exists a transition $\langle q, a, \mu, G, A, q'\rangle$ in the IOSTS $\mathcal{I}$ and the guard G of this transition evaluates to $true$ with the given valuations v and w. The new variable valuation v' of the system after firing the transition is calculated by applying the assignments in A for each variable, i.e., for each variable $x \in V$, the right-hand side of the variable's assignment A^x is evaluated according to v and w and assigned to x in the new valuation v'. The system parameters do not change.

The IOSTS transition relation $\Rightarrow$ is a tuple $\langle s, l, s'\rangle$ whereas $s, s' \in S$ and $l \in (\Upsilon \cup \Omega)^*$ and can be derived from $\rightarrow$ by dropping all internal actions. $\Rightarrow$ is defined by Rusu et al. [67] through the following three rules:

1. $s \overset{\epsilon}{\Rightarrow} s' \ =_{def} \ \exists \tau_1, ..., \tau_n \in \Sigma^{int}, \exists s_1, ..., s_{n-1} \in S, s \xrightarrow{\tau_1} s_1 ... s_{n-1} \xrightarrow{\tau_n} s'$
   $s \overset{\epsilon}{\Rightarrow} s'$ means that there is a sequence of transitions labelled with internal actions $(\tau_1, ..., \tau_n)$ which can be fired to get from state s to state s' via several intermediate states $(s_1, ..., s_{n-1})$. In other words, there is a way from s to s' without observable behaviour.
2. $s \overset{\alpha}{\Rightarrow} s' \ =_{def} \ \exists s_1, s_2 \in S, s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{\alpha} s_2 \overset{\epsilon}{\Rightarrow} s'$ with $\alpha \in \Upsilon \cup \Omega$
   $s \overset{\alpha}{\Rightarrow} s'$ means that there is a way to get from state s to state s' by following a sequence of transitions consisting of one transition labelled with the valued input/output α preceded and/or followed by a finite number of transitions labelled with an internal action.
3. $s \overset{\sigma}{\Rightarrow} s' \ =_{def} \ \exists s_1, ..., s_{n-1} \in S, s \overset{\alpha_1}{\Rightarrow} s_1 ... s_{n-1} \overset{\alpha_n}{\Rightarrow} s'$ with $\sigma = \alpha_1, ..., \alpha_n \in (\Upsilon \cup \Omega)^n$ with $n > 1$
   $s \overset{\sigma}{\Rightarrow} s'$ says that there is a way from state s to s' by firing transitions labelled with the valued inputs/outputs from σ. Before, between and after these transitions, there can be an infinite number of internal actions.

The following terminology defined by Rusu et al. [67] is useful when talking about an IOSTS $\mathcal{I} = \langle D, \Theta, Q, q_0, \Sigma, \mathcal{T}\rangle$ with semantics given by the LTS $\mathcal{L} = \langle S, S_0, L, \rightarrow\rangle$:

traces $traces(\mathcal{I}) = \left\{\sigma \in (\Upsilon \cup \Omega)^* \;\middle|\; \exists s_0 \in S_0, \exists s \in S : s_0 \overset{\sigma}{\Rightarrow} s\right\}$
The traces of an IOSTS $\mathcal{I}$ are all sequences of valued inputs/outputs which lead from an initial state s_0 to any other state s of the IOSTS.

after $\mathcal{I}$ *after* $\sigma = \left\{ s \in S \mid \exists s_0 \in S_0 : s_0 \overset{\sigma}{\Rightarrow} s \right\}$
$\mathcal{I}$ *after* σ denotes the set of states which can be reached from an initial state s_0 following the trace σ.

out $out(S') = \left\{ \alpha \in \Omega \mid \exists s' \in S', \exists s \in S : s' \overset{\alpha}{\Rightarrow} s \right\}$ with $S' \subseteq S$
$out(S')$ denotes the set of valued outputs observable in the states contained in S'. These observations can possibly take place after internal actions.

pref *pref*(L) is defined as the set of strict prefixes of sequences in L, where L is a set of traces.

2.2.4 Types of IOSTS

According to Rusu et al. [67], an IOSTS can have several attributes:

Instantiated Given an IOSTS $\mathcal{I}$ with system parameters P and a valuation of these system parameters $\pi \in Val(P)$, an instance of $\mathcal{I}$ named $\mathcal{I}(\pi)$ is obtained by substituting each system parameter $p \in P$ by its value $\pi(p)$.

Initialized If the initial condition Θ assigns exactly one value to each variable $v \in V$, then an instantiated IOSTS is called initialized. If all instances of an arbitrary IOSTS are initialized, the IOSTS is initialized.

Deterministic An IOSTS is deterministic if the following two conditions hold:

1. $\forall s \in S : \left| \cup_{\tau \in \Sigma^{int}} \left\{ s' \in S \mid s \xrightarrow{\tau} s' \right\} \right| \leq 1$
   The next state s' depends only on the current state s when executing an internal action.
2. $\forall s \in S,\ \forall \alpha \in \Upsilon \cup \Omega : \left| \left\{ s' \in S \mid s \xrightarrow{\alpha} s' \right\} \right| \leq 1$
   If a valued input/output α is executed, the next state s' depends only on the current state s and the valued input/output α.

Complete An IOSTS location q is complete if the following condition holds:
$\forall s \in S,\ \forall \alpha \in \mathit{\Upsilon} \cup \mathit{\Omega} \cup \Sigma^{int} : \left\{ s' \in S \mid s \overset{\alpha}{\Rightarrow} s' \right\} \neq \emptyset$
That is, for all states s and all valued inputs/outputs or internal actions α, the set of successor states reachable through α must be nonempty. An IOSTS is complete if all its locations are complete. An IOSTS is input-complete if the above condition holds for all valued inputs $\alpha \in \mathit{\Upsilon}$.

2.3 Conformance Testing with IOSTS

The general aspects of conformance testing have already been discussed in Section 2.1. According to Rusu et al. [67], conformance testing with IOSTS means testing a conformance relation between a formal specification of the system in the form of an IOSTS and an implementation of the system. Test purposes define what will be tested throughout a particular test case. During testing, verdicts about the conformance are returned. In the following, more detailed aspects of conformance testing based on IOSTS will be considered.

2.3.1 Specification

According to Rusu et al. [67], a specification is a formal model of a system. It is an initialized IOSTS, which describes how the system should act. Hence, IOSTS belong to the set of behavioural specifications (see Section 2.1). For the already mentioned Triangle Type Checker (see Section 2.2.2), the specification IOSTS is depicted in Figure 2.2.

2.3.2 Implementation

The implementation under test is a black-box application. Due to the test hypothesis (see Section 2.1), which assumes that each implementation can be represented by a model, the implementation can be considered as an IOSTS. The only thing known for sure about the implementation's IOSTS is its interface, which is assumed to be same as the specification's interface [67]. Hence, the following items of the implementation's IOSTS are known:

- The input alphabet Σ^i (input actions).
- The output alphabet Σ^o (output actions).
- The signatures $sig(a)$ of all actions $a \in \Sigma^i \cup \Sigma^o$.

The Triangle Type Checker implementation for the running example is a single Java file, which is included in the archive `stg_backend.tgz` (`/EssaiTri/triangle.java`). It can be downloaded from the download section of the STG web page[4].

2.3.3 Test Purpose

As already explained in Section 2.1, a test purpose describes what shall be tested. When working with STG, a test purpose $\mathcal{TP}$ for a specification $\mathcal{S}$ is an initialized and complete IOSTS, which is compatible with $\mathcal{S}$ regarding the product operation (see Section 3.1.2). It selects a part of the specification that shall be tested. Its set of locations $Q_{\mathcal{TP}}$ includes at least one *Accept* location. *Accept* locations indicate behaviours of the specification which shall be tested. They must not have outgoing transitions that lead to other locations. Transitions leading to *Accept* locations (except self-loops) must not be labelled by internal actions. Test purposes may also contain *Reject* locations. They indicate behaviours that are not targeted by the test purpose [80].

A good test purpose should be simple and much smaller than the specification itself. The design of test purposes has to be done by hand and is said to be an iterative process [67; 80]. Figure 2.3 was generated by STG. It shows a possible test purpose for the specification depicted in Figure 2.2. It tests whether the Triangle Type Checker correctly identifies invalid triangles. Since the specification itself is rather small, the test purpose is very simple.

Test purposes have to be complete by definition. For small IOSTS like the one depicted in Figure 2.3, it would be easy to keep an overview and thus to ensure completeness by hand. For bigger test purposes, ensuring completeness manually is troublesome. For this reason, STG does not require complete test purposes from the user. It is able to compute missing transitions to complete an IOSTS. The completion of an IOSTS will be elaborated in Section 3.1.1.

2.3.4 Test Case

Rusu et al. [67] define test cases to be initialized, deterministic IOSTS without internal actions in order to react promptly to inputs from the implementation. If an IOSTS $\mathcal{TC}$ has these three attributes, then the following condition holds: $\forall \sigma \in traces(\mathcal{TC}) : |\mathcal{TC} \textit{ after } \sigma| = 1$. This means that each trace of the test case leads to exactly one state. Hence, if the same trace is executed several times on a given implementation, the test case always produces the same verdict. Nevertheless, one test case can result in different verdicts for the same implementation. This can happen, because the implementation may be non-deterministic, i.e., it may choose to follow different traces [80].

A further requirement for proper test cases is the input-completeness of all locations of a test case. Locations which are included in three special sets of locations are an exception. These three sets are *Pass*, *Fail*, and *Inconclusive*. They are pairwise disjoint and used to generate verdicts (cf. Section 2.1).

[4]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)

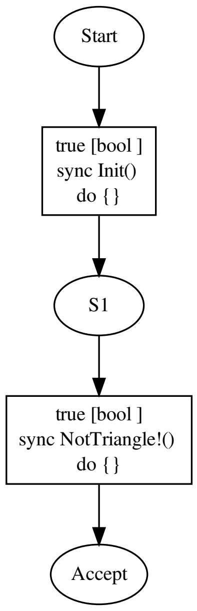


Figure 2.3: This test purpose for the Triangle Type Checker specification checks whether the system correctly identifies invalid triangles.

To determine the actual verdict, the end location of the test case after running the test case in parallel to the implementation is used. If the end location is part of the set *Pass*, the test case passed - the implementation worked correctly regarding the specification. If the end location is in the set *Fail*, the test case detected an error - the implementation is not conform to the specification. In the case of an end location in the set *Inconclusive*, the implementation worked correctly, but the test goal defined in the test purpose could not been reached because of allowed non-determinism in the implementation.

Figure 2.4 shows the test case generated with STG from the specification depicted in Figure 2.2 and the test purpose of Figure 2.3. The guards of the transitions have been rewritten and/or combined during test case generation (see Section 3.1.5). The location *End_Accept* belongs to the set *Pass*. The location *inconc* is contained in the set *Inconclusive*, since it indicates valid behaviour but does not lead to an *Accept* state any more. To keep the test case as simple as possible, no location of the set *Fail* is depicted. STG automatically generates a *Fail* verdict if an input from the implementation does not match any transition leaving the current state of the test case.

The process of generating such a test case from a specification and a test purpose will be explained in more detail in Section 3.1. The most important operation for the creation of test cases is the so-called product operation (denoted $\times$), which is applied on a specification and a test purpose. This particular operation will be discussed in Section 3.1.2.

2.3.5 Conformance Relations

An implementation under test is linked with instances of the specification and the test purpose via conformance relations. Rusu et al. [67] as well as Zinovieva-Leroux [80] formally define two conformance relations for IOSTS :

conf The conformance relation $\mathcal{I}(\pi)$ *conf* $\mathcal{S}(\pi)$ expresses that an implementation modelled by the instantiated IOSTS $\mathcal{I}(\pi)$ conforms to the instance $\mathcal{S}(\pi)$ of the specification $\mathcal{S}$. Its definition is:

$$\mathcal{I}(\pi)\ \mathit{conf}\ \mathcal{S}(\pi) \quad =_{def} \quad \forall \sigma \in traces(\mathcal{S}(\pi)) : out(\mathcal{I}(\pi)\ after\ \sigma) \subseteq out(\mathcal{S}(\pi)\ after\ \sigma)$$

That is, after each trace of the specification, all possible valued outputs of the implementation are included in the set of valued outputs of the specification.

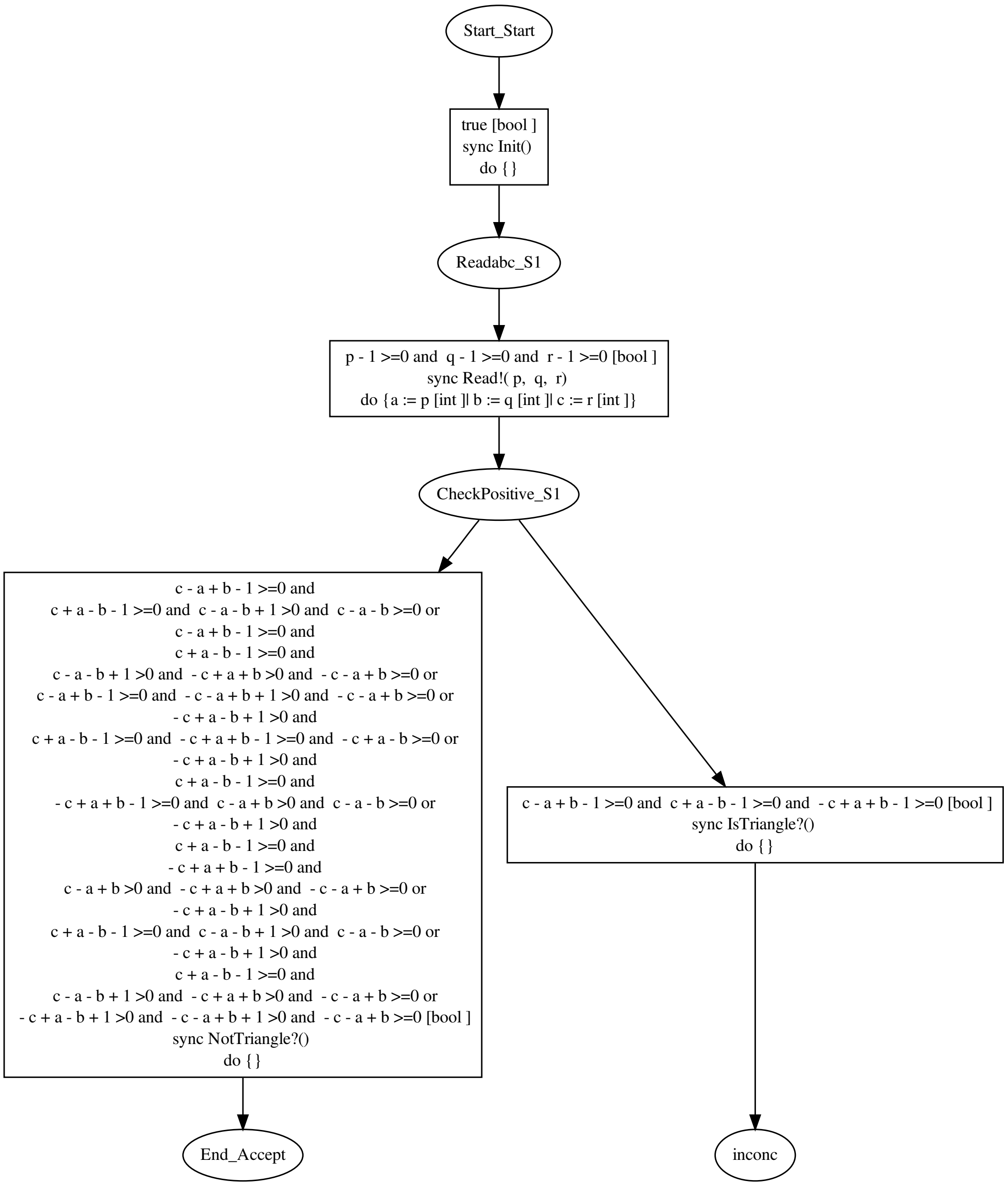

Start_Start
true [bool]
sync Init()
do { }
Readabc_S1
p - 1 >=0 and q - 1 >=0 and r - 1 >=0 [bool]
sync Read!(p, q, r)
do {a := p [int]| b := q [int]| c := r [int]}
CheckPositive_S1
c - a + b - 1 >=0 and
c + a - b - 1 >=0 and c - a - b + 1 >0 and c - a - b >=0 or
c - a + b - 1 >=0 and
c + a - b - 1 >=0 and
c - a - b + 1 >0 and - c + a + b >0 and - c - a + b >=0 or
c - a + b - 1 >=0 and - c - a + b + 1 >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and - c + a + b - 1 >=0 and - c + a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and c - a + b >0 and c - a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and
c - a + b >0 and - c + a + b >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and c - a - b + 1 >0 and c - a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
c - a - b + 1 >0 and - c + a + b >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and - c - a + b + 1 >0 and - c - a + b >=0 [bool]
sync NotTriangle?()
do { }
c - a + b - 1 >=0 and c + a - b - 1 >=0 and - c + a + b - 1 >=0 [bool]
sync IsTriangle?()
do { }
End_Accept
inconc

Figure 2.4: The test case generated by STG from the specification depicted in Figure 2.2 and the test purpose shown in Figure 2.3. *End_Accept* belongs to the set *Pass*, *inconc* to the set *Inconclusive*. To simplify matters, no location of the set *Fail* is depicted.

According to Zinovieva-Leroux [80], this conformance relation can be extended to IOSTS that are not instantiated. Therefor, in addition to the test hypothesis that states that each implementation can be modelled, a new hypothesis has to be introduced. It says that there exists a one-to-one correspondence between the system parameters of an implementation and the system parameters of a specification. Thereby, conformance of not instantiated IOSTS can be defined:

$$\mathcal{I}\ \mathit{conf}\ \mathcal{S} \quad =_{def} \quad \forall v \in Val(P) : \big(\ \mathcal{I}(v)\ conf\ \mathcal{S}(v)\ \big)$$

This means that an implementation, which is modelled by the IOSTS $\mathcal{I}$, conforms to a specification $\mathcal{S}$ if for all possible valuations $v \in Val(P)$ of their system parameters P, the implementation instantiated with v conforms to the specification instantiated with the same valuation.

conf links implementations and specifications. In order to take into account test purposes, which are the third component of STG's testing theory, the conformance relation $\mathit{conf}_{\mathcal{TP}}$ is defined as follows:

conf$_{\mathcal{TP}}$ The conformance relation $\mathcal{I}(\pi)$ $\mathit{conf}_{\mathcal{TP}}$ $\mathcal{S}(\pi)$ means that the implementation, which is modelled by the instantiated IOSTS $\mathcal{I}(\pi)$, conforms to the instance $\mathcal{S}(\pi)$ of the specification $\mathcal{S}$ and relative to the instantiated test purpose $\mathcal{TP}$. Its definition is:

$$\mathcal{I}(\pi)\ \mathit{conf}_{\mathcal{TP}}\ \mathcal{S}(\pi)\ =_{def}\ \forall \sigma \in \mathit{pref}(Atraces(\mathcal{P}(\pi))) : out(\mathcal{I}(\pi)\ \mathit{after}\ \sigma) \subseteq out(\mathcal{S}(\pi)\ \mathit{after}\ \sigma)$$

$\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ denotes the product of $\mathcal{S}$ and $\mathcal{TP}$, $\mathcal{P}(\pi) = (\mathcal{S} \times \mathcal{TP})(\pi)$ is an instance of this product. $Atraces(\mathcal{P})$ is the set of traces σ, which lead from $\mathcal{P}$'s initial state to a state with a location $q \in Q_{\mathcal{S}} \times Accept_{\mathcal{TP}}$, whereby $Q_{\mathcal{S}}$ is the set of states in $\mathcal{S}$ and $Accept_{\mathcal{TP}}$ is the set of *Accept* states in the test purpose $\mathcal{TP}$. $Atraces(\mathcal{P}) \subseteq traces(\mathcal{S})$ applies.

In other words, $\mathit{conf}_{\mathcal{TP}}$ means that after each strict prefix of a selected trace of the specification, all possible valued outputs of the implementation have to be included in the set of valued outputs of the specification.

Similar to *conf*, $\mathit{conf}_{\mathcal{TP}}$ can be generalized to be applicable to uninstantiated IOSTS [80]:

$$\mathcal{I}\ \mathit{conf}_{\mathcal{TP}}\ \mathcal{S}\ =_{def}\ \forall v \in Val(P_{\mathcal{S}} \cup P_{\mathcal{TP}}) : \Big(\ \mathcal{I}(v \downarrow P_{\mathcal{I}})\ \ \mathit{conf}_{\mathcal{TP}(v\downarrow P_{\mathcal{TP}})}\ \ \mathcal{S}(v \downarrow P_{\mathcal{S}})\ \Big)$$

In this definition, the system parameters of the specification $\mathcal{S}$ are denoted by $P_{\mathcal{S}}$, the system parameters of the test purpose $\mathcal{TP}$ are denoted by $P_{\mathcal{TP}}$, and the system parameters of the implementation $\mathcal{I}$ are denoted by $P_{\mathcal{I}}$. Note that the elements in $P_{\mathcal{I}}$ are assumed to be one-to-one correspondent to the elements in $P_{\mathcal{S}}$. Expressions of the form $v \downarrow D'$ with $v \in Val(D)$ and $D' \subseteq D$ denote so-called *projections*. They return the values $v(d)$ for each data item $d \in D'$.

Hence, an implementation, which is modelled by the IOSTS $\mathcal{I}$, conforms to a specification $\mathcal{S}$ relative to a test purpose $\mathcal{TP}$ if for all possible valuations $v \in Val(P_{\mathcal{S}} \cup P_{\mathcal{TP}})$, the implementation instantiated with $v \downarrow P_{\mathcal{I}}$ and the specification instantiated with $v \downarrow P_{\mathcal{S}}$ are conform relative to the test purpose instantiated with $v \downarrow P_{\mathcal{TP}}$.

conf and $\mathit{conf}_{\mathcal{TP}}$ are slightly different. *conf* is stronger and therefore it implies $\mathit{conf}_{\mathcal{TP}}$. If an implementation conforms to a specification, then it conforms to the specification relative to all test purposes for this specification. Conversely, $\mathit{conf}_{\mathcal{TP}}$ does not imply *conf* [80]. Since STG relies on a set of manually designed test purposes, only $\mathit{conf}_{\mathcal{TP}}$ can be verified for each specified test purpose. The user is responsible for providing a sufficient number of test purposes to achieve conformance according to the *conf* relation.

Conformance Relations and Quiescence

According to Zinovieva-Leroux [80], the above defined conformance relations *conf* and $\mathit{conf}_{\mathcal{TP}}$ are a weaker version of the conformance relations *ioco* and *ioconf* defined by Tretmans [75; 76]. The difference lies within quiescence (outputlocks, deadlocks, livelocks). *ioco* and *ioconf* are based on LTS (Labelled Transition Systems [51]) and take into account quiescence. Quiescence is not considered by *conf* and $\mathit{conf}_{\mathcal{TP}}$, which are defined for IOSTS. Note that deciding whether an IOSTS is quiescent or not is undecidable in general [80].

3 Existing Approach using Reachability Analysis

An already existing approach for symbolic test case generation was implemented in the prototype tool STG, shorthand for Symbolic Test Generator. It was designed to be applicable to reactive systems (see Section 1.3.1) specified as IOSTS (Input Output Symbolic Transition Systems, see Section 2.2). STG was developed at IRISA/INRIA[1] Rennes (France) in the course of the project *VerTeCs*[2] [25; 49].

Figure 3.1 was inspired by Clarke et al. [26] and gives an overview of STG's work flow, which corresponds to the model-based testing process presented in Section 1.2.1. The main steps are:

1. System modelling and design of test purposes:
   The specification of the system under test has to be modelled as an IOSTS. Test purposes have to be designed and expressed in IOSTS syntax. They serve as test case specifications and select the part of the system which should be tested. Both tasks have to be done manually by the user.
2. Test generation:
   STG processes the two user-defined IOSTS (specification and test purpose) to generate an IOSTS test case. This abstract test case will be converted into Java format to generate a concrete test case.
3. Test execution:
   For test execution, the implementation under test is executed in parallel to the generated test case, which gives input to the implementation and checks the returned output. The result is a verdict, which can be *pass*, *fail*, or *inconclusive*.

Two procedures of the above outlined workflow are automated by STG: test generation and test execution. In the following, those two work steps will be described in more detail. Furthermore, a short description of the installation and execution of STG's tool framework is provided. Finally, problems of the STG prototype will be discussed.

3.1 Test Case Generation with STG

The process performed by STG to generate a test case from a given specification and test purpose is:

1. The test purpose is made complete, i.e., missing transitions are added.
2. The specification and the complete test purpose are processed by the product operation to generate the basis for the resulting test case, which will be called product in the following.
3. Then the product is closed and determinized to become a valid test case.
4. The closed and determinized product is simplified via reachability and coreachability analyses.

Figure 3.2 illustrates STG's process for test case generation, which has been briefly introduced above. Framed boxes depict the kinds of objects involved in the procedures (IOSTS process, AUTO file, NBAC file, ...). The dark-red labels near these boxes describe the names given to these objects by STG within the test case generation process, whereas *basename* stands for the name of the STG specification file (`basename.stg`). Arrows and their captions represent the operations, which need to be performed on their source object(s) to obtain the target object. Not every operation is implemented in STG itself. For the reachability and coreachability analysis phases, the external tool NBac was used (see Section 3.3.1). The operations implemented in NBac and used by STG for test case generation are: (1) auto2nbac, (2) nbac reachability analysis, (3) nbac coreachability analysis, and (4) nbac2auto. The following sections will describe the different phases of STG's test case generation procedure in more detail.

[1]`http://www.irisa.fr` (last visit 2009-09-27)
[2]`http://ralyx.inria.fr/2006/Raweb/vertecs/uid34.html` (last visit 2009-09-27)

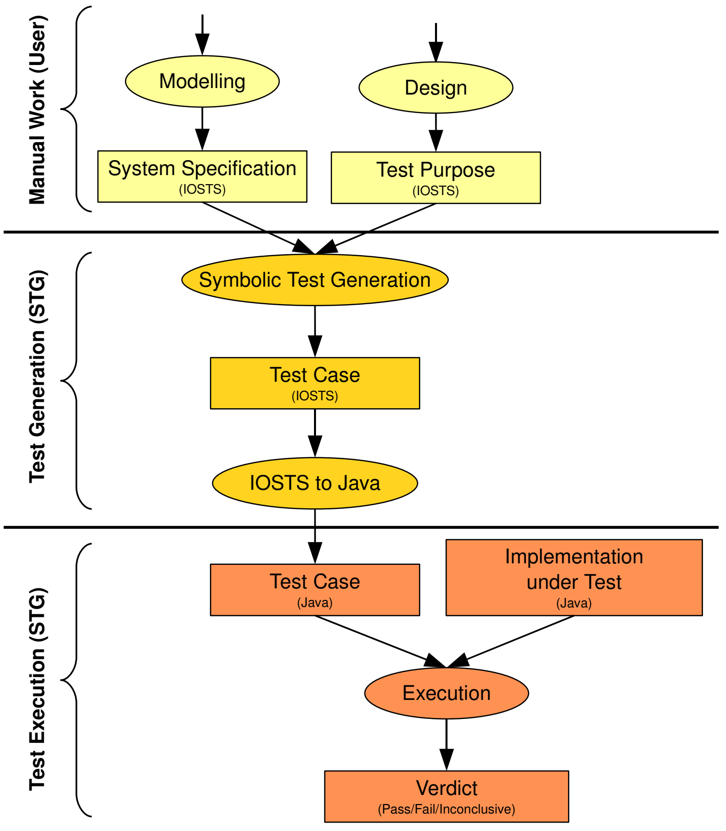


Figure 3.1: STG's work flow consists of three main steps: system modelling and test purpose design, test generation, and test execution.

3.1.1 Completion

STG allows users to design test purposes which are not complete (see Section 2.3.3). Since test purposes are required to be complete in order to generate test cases, STG has to add transitions to achieve completeness of a test purpose. According to Zinovieva-Leroux [80], a test purpose $\mathcal{TP}$ is made complete with respect to its specification $\mathcal{S}$ by application of the following rules:

1. If a location q of $\mathcal{TP}$ has no outgoing transitions labelled with an action a of $\mathcal{S}$, it is assumed that the test purpose designer does not care about the presence of this action in the implementation under test (IUT) at this location. Consequently, a self-loop labelled by a will be added to q.
2. If a location q of $\mathcal{TP}$ has outgoing transitions labelled with an action a of $\mathcal{S}$, two cases have to be distinguished:
   (a) The test purpose designer wants to test the presence of the action a in the IUT under some condition G. In other words, the transition labelled with a and guard G does not lead to the *Reject* location. In this case, it is assumed that the user does not want to test the presence of a under other conditions than G. Consequently, a new transition leading from q to *Reject* labelled with a and having the guard $\neg G$ will be added.
   (b) The test purpose designer does not want to test the presence of the action a in the IUT under a certain condition G, i.e., the transition labelled with a and G leads to a *Reject* location. In this case, it is assumed that the user wants to test the presence of a in the implementation under any other condition. Consequently, a self-loop labelled with a and $\neg G$ will be added to q.

For each of the transitions added during completion of the test purpose, the set of assignments is the so-called set of identity assignments. This means that each variable of $\mathcal{TP}$ is assigned to itself.

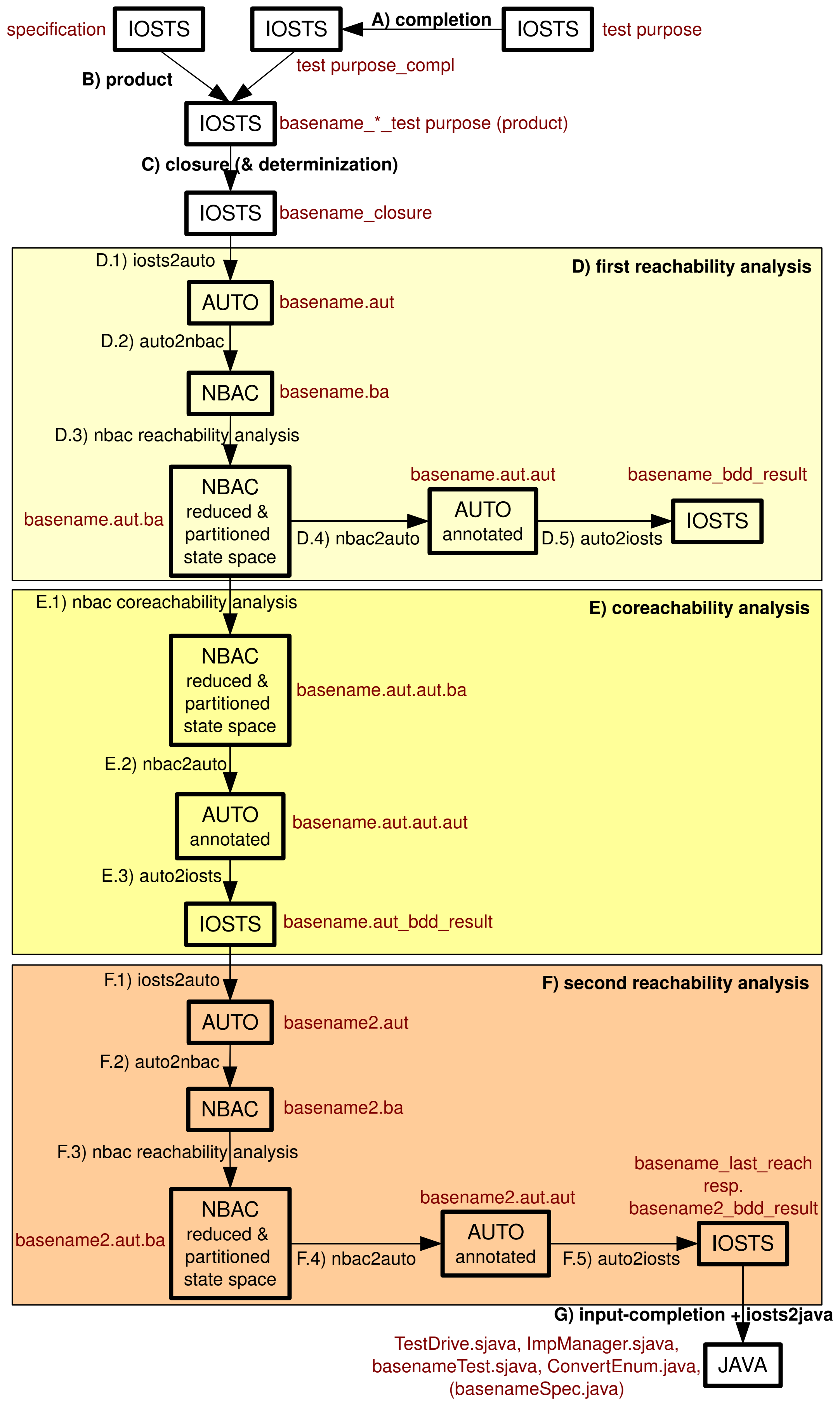
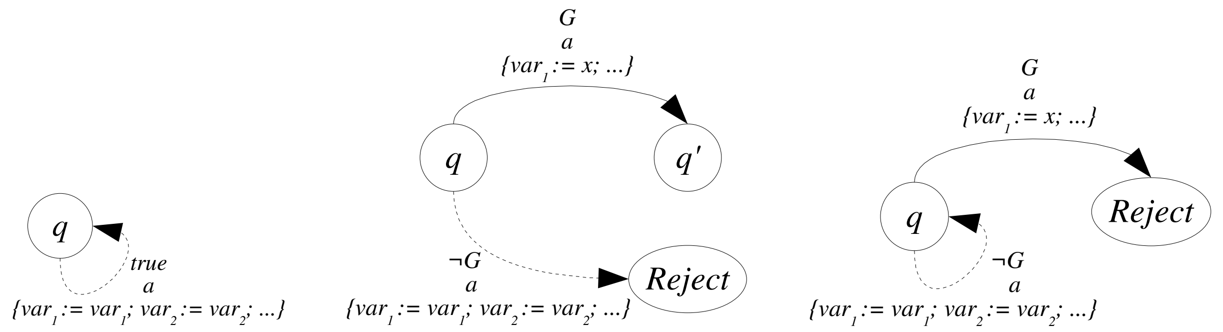

specification
IOSTS
IOSTS
A) completion
IOSTS
test purpose
test purpose_compl
B) product
IOSTS
basename_*_test purpose (product)
C) closure (& determinization)
IOSTS
basename_closure
D) first reachability analysis
D.1) iosts2auto
AUTO
basename.aut
D.2) auto2nbac
NBAC
basename.ba
D.3) nbac reachability analysis
NBAC reduced & partitioned state space
basename.aut.ba
D.4) nbac2auto
basename.aut.aut
AUTO annotated
D.5) auto2iosts
basename_bdd_result
IOSTS
E) coreachability analysis
E.1) nbac coreachability analysis
NBAC reduced & partitioned state space
basename.aut.aut.ba
E.2) nbac2auto
AUTO annotated
basename.aut.aut.aut
E.3) auto2iosts
IOSTS
basename.aut_bdd_result
F) second reachability analysis
F.1) iosts2auto
AUTO
basename2.aut
F.2) auto2nbac
NBAC
basename2.ba
F.3) nbac reachability analysis
NBAC reduced & partitioned state space
basename2.aut.ba
F.4) nbac2auto
basename2.aut.aut
AUTO annotated
F.5) auto2iosts
basename_last_reach resp. basename2_bdd_result
IOSTS
G) input-completion + iosts2java
TestDrive.sjava, ImpManager.sjava, basenameTest.sjava, ConvertEnum.java, (basenameSpec.java)
JAVA

Figure 3.2: STG's test case generation process.


q
true
a
{var1:= var1; var2:= var2; ...}
G
a
{var1:= x; ...}
q
q'
¬G
a
{var1:= var1; var2:= var2; ...}
Reject
G
a
{var1:= x; ...}
q
Reject
¬G
a
{var1:= var1; var2:= var2; ...}

(a) Rule 1. There are no transitions labelled by a and starting at location q. A self-loop labelled with a will be added to q.

(b) Rule 2.(a) The transition with action a and guard G does not lead to *Reject*. A transition leading to *Reject* with a and guard $\neg G$ will be added.

(c) Rule 2.(b) The transition with action a and guard G leads to *Reject*. A self-loop with a and guard $\neg G$ will be added.

Figure 3.3: The rules for completing Input Output Symbolic Transition Systems. Regardless of which rule has to be applied, the set of assignments is the set of identity assignments.

Figure 3.3 depicts the three rules for the completion of IOSTS. Continuous lines mark transitions which have already been in the IOSTS before applying a completion rule. Dashed lines depict transitions which are added through the application of a completion rule.

STG's approach for completion differs slightly from the rules above. For a direct demonstration of the differences, the same example as in the work of Zinovieva-Leroux [80] will be used. Figure 3.4 shows the completed test purpose, which is intended to test the coffee functionality of a vending machine for hot beverages. Black continuous lines depict the incomplete test purpose, which has been designed by the user. Dashed lines represent the transitions added during completion. Grey parts of the IOSTS are only added by the algorithm described by Zinovieva-Leroux [80]. STG does not add self-loops in *Accept* or *Reject* locations. It also does not add transitions labelled with actions which are only defined in the specification but not in the test purpose itself. Since the actions *ChooseBeverage*, *Coin*, and the internal action *tau* are not used in the test purpose, STG does not add transitions for them during test purpose completion. However, STG is implemented correctly and the missing transitions are added implicitly when the actions defined in the specification $\mathcal{S}$ are known. That is, when the product (see Section 3.1.2) between the specification $\mathcal{S}$ and the test purpose $\mathcal{TP}_{comp}$, which is complete in terms of STG, is calculated.

Example: Triangle Type Checker

To illustrate the completion mechanism on a practical example, the Triangle Type Checker specification as introduced in Section 2.2.2 with the test purpose depicted in Figure 2.3 will be used. Since the specification itself is rather small, the used test purpose is very simple and does not change through STG's completion. However, during the product calculation, it will be entirely completed, i.e., it will be completed with respect to its specification.

The resulting IOSTS is depicted in Figure 3.5. Seven transitions have been added to the location *Start*, one for each action defined in the specification and not labelling a transition starting at *Start*. Since the action *NotTriangle* is already labelling a transition originating from location *S1*, only six transitions have been added starting at *S1*: one for each action defined in the specification except for *NotTriangle*. It is not specified if the *Accept* location is completed as well.

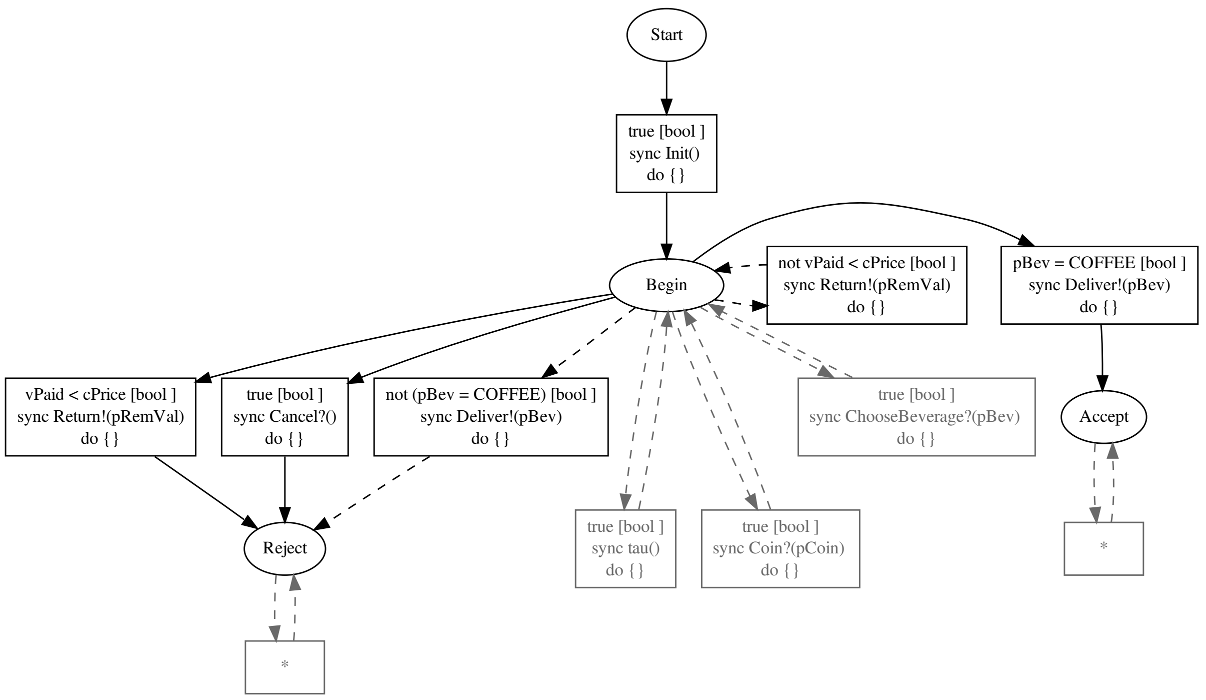


Figure 3.4: The completion of a test purpose, which is intended to test the coffee functionality of a vending machine for hot beverages. Black continuous lines depict the incomplete test purpose designed by the tester. Dashed lines represent the completing transitions. Grey parts are added by Zinovieva-Leroux's algorithm [80], but not during STG's completion process.

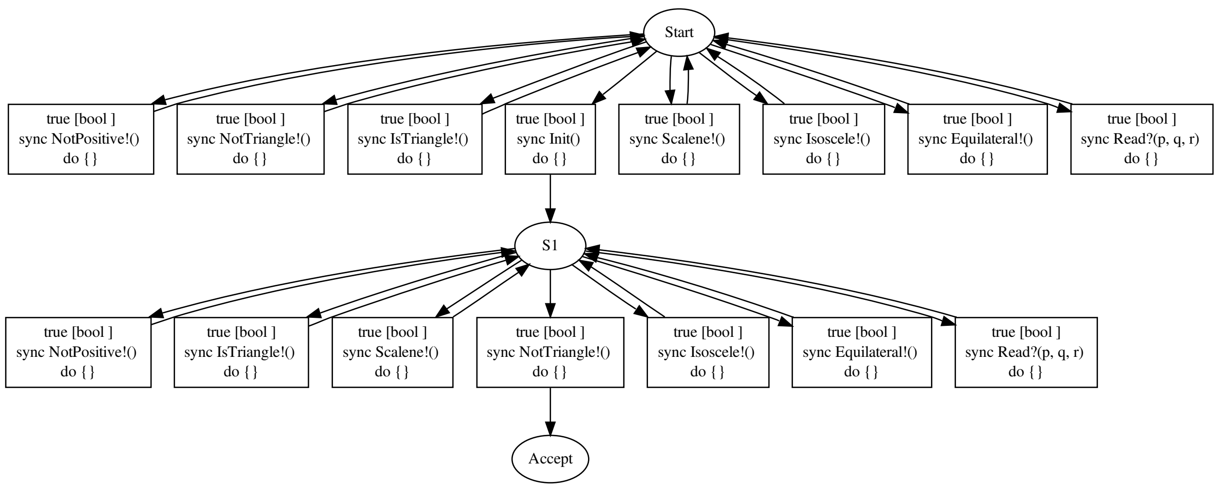


Figure 3.5: The test purpose shown in Figure 2.3, which has been completed with respect to its specification shown in Figure 2.2.

3.1.2 Product

According to Zinovieva-Leroux [80], the product operation is applied on two IOSTS: the system specification and the completed test purpose. It is performed to find out which behaviours of the specification are accepted by the test purpose. The product operation will be denoted as $\times$ in the following and can only be applied if the two IOSTS operands meet certain compatibility requirements. These requirements as well as the product operation itself will be defined below and an illustration using the Triangle Type Checker example will be given.

Compatible for Product

Two IOSTS $\mathcal{I}_1$ and $\mathcal{I}_2$ with data sets $D_1 = V_1 \cup P_1 \cup M_1$ and $D_2 = V_2 \cup P_2 \cup M_2$ and alphabets $\Sigma_1 = \Sigma_1^i \cup \Sigma_1^o \cup \Sigma_1^{int}$ and $\Sigma_2 = \Sigma_2^i \cup \Sigma_2^o \cup \Sigma_2^{int}$ are compatible for product if the following conditions hold [80]:

- $\mathcal{I}_1$ and $\mathcal{I}_2$ must not have shared variables or action messages ($V_1 \cap V_2 = \emptyset$ and $M_1 \cap M_2 = \emptyset$). Common system parameters are no problem. It is also allowed that the variables of $\mathcal{I}_1$ serve as system parameters of $\mathcal{I}_2$ and vice versa. The only thing to consider in the case of common data is that they must be of the same type in both IOSTS.
- $\mathcal{I}_1$ and $\mathcal{I}_2$ must have exactly the same alphabet for inputs, outputs, and internal actions ($\Sigma_1^i = \Sigma_2^i$, $\Sigma_1^o = \Sigma_2^o$, and $\Sigma_1^{int} = \Sigma_2^{int}$). The signatures of common actions must be the same in both IOSTS.

Product Operation

The product of two IOSTS $\mathcal{I}_1 = \langle D_1, \Theta_1, Q_1, q_{01}, \Sigma_1, \mathcal{T}_1 \rangle$ and $\mathcal{I}_2 = \langle D_2, \Theta_2, Q_2, q_{02}, \Sigma_2, \mathcal{T}_2 \rangle$, which are compatible for product, is an IOSTS $\mathcal{P} = \mathcal{I}_1 \times \mathcal{I}_2$ having the following attributes [80]:

- $D = V \cup P \cup M$ with $V = V_1 \cup V_2$, $P = (P_1 \cup P_2) \setminus (V_1 \cup V_2)$, and $M = M_1$
- $\Theta = \Theta_1 \wedge \Theta_2$
- $Q = Q_1 \times Q_2$
- $q_0 = \langle q_{01}, q_{02} \rangle \in Q$
- $\Sigma = \Sigma^i \cup \Sigma^o \cup \Sigma^{int}$ with $\Sigma^i = \Sigma_1^i = \Sigma_2^i$, $\Sigma^o = \Sigma_1^o = \Sigma_2^o$, and $\Sigma^{int} = \Sigma_1^{int} = \Sigma_2^{int}$
- $\mathcal{T}$ is calculated from $\mathcal{T}_1$ and $\mathcal{T}_2$:
  A new symbolic transition $t \in \mathcal{T}$ is created for two symbolic transitions $t_1 \in \mathcal{T}_1$ and $t_2 \in \mathcal{T}_2$, which are both labelled by the same action $a \in (\Sigma_1 = \Sigma_2)$. t is obtained according to the following inference rule:

$$\frac{\begin{array}{c}\langle q_1, a, \mu_1, G_1, A_1, q_1' \rangle \in \mathcal{T}_1 \\ \langle q_2, a, \mu_2, G_2, A_2, q_2' \rangle \in \mathcal{T}_2 \\ a \in (\Sigma_1 = \Sigma_2)\end{array}}{\langle \langle q_1, q_2 \rangle, a, \mu_1, G_1 \wedge G_2[\mu_2/\mu_1], A_1 \cup A_2[\mu_2/\mu_1], \langle q_1', q_2' \rangle \rangle \in \mathcal{T}}$$

  $G_2[\mu_2/\mu_1]$ denotes the guard of t_2 in which each message $m_2^i \in \mu_2$ of t_2's action a is substituted by the corresponding message $m_1^i \in \mu_1$. $A_2[\mu_2/\mu_1]$ denotes the set of assignments with the same replacement of messages.

It is important to say that the STG prototype additionally inverts input to output and output to input actions in the resulting product. Since the implementation has the same interface as the specification, the product has to change its inputs to outputs and its outputs to inputs. This is necessary for the communication between the test case (which is based on the product) and the implementation under test.

While using STG to generate test cases for the Session Initiation Protocol (SIP) Registrar (see Section 6.2) and the Conference Protocol (see Section 6.3), it has been observed that the product of the specification and the test purpose contains a large number of duplicate transitions. This redundancy does not affect the correctness of the resulting product, but it could influence the performance of the remaining steps of the test case generation.

Example: Triangle Type Checker

Again, the Triangle Type Checker example will be used for illustration of STG's product calculation. The IOSTS shown in Figure 2.2 will be denoted by $\mathcal{S} = \langle D_S, \Theta_S, Q_S, q_{0S}, \Sigma_S, \mathcal{T}_S \rangle$, the IOSTS depicted in Figure 3.5 by $\mathcal{TP}_{comp} = \langle D_{TP_{comp}}, \Theta_{TP_{comp}}, Q_{TP_{comp}}, q_{0TP_{comp}}, \Sigma_{TP_{comp}}, \mathcal{T}_{TP_{comp}} \rangle$. $\mathcal{S}$ and $\mathcal{TP}_{comp}$ are compatible for product since they do not have shared variables nor shared action messages. Their alphabets are compatible since all common actions have the same signature in both IOSTS.

The product $\mathcal{P} = \mathcal{S} \times \mathcal{TP}_{comp}$ generated by STG is shown in Figure 3.6. It is calculated in the following way:

- $V = V_S \cup V_{TP_{comp}} = \{a, b, c\} \cup \emptyset = \{a, b, c\}$
  $P = (P_S \cup P_{TP_{comp}}) \setminus (V_S \cup V_{TP_{comp}}) = (\emptyset \cup \emptyset) \setminus (\{a, b, c\} \cup \emptyset) = \emptyset$
  $M = M_S = \{p, q, r\}$
  $D = V \cup P \cup M = \{a, b, c\} \cup \emptyset \cup \{p, q, r\} = \{a, b, c, p, q, r\}$: The data set is equal to the data set of the IOSTS $\mathcal{S}$.
- $\Theta = \Theta_S \wedge \Theta_{TP_{comp}} = true \wedge true = true$: The initial condition is $true$.
- $q_0 = \langle q_{0S}, q_{0TP_{comp}} \rangle = \langle Start, Start \rangle$: The initial location is the pair consisting of the two initial locations.
- The alphabets of input, output, and internal actions are the same as the ones of $\mathcal{S}$.
- The symbolic transitions of $\mathcal{P}$ are computed in the following way:
  - Starting from the initial location, both IOSTS can execute a transition labelled with *Init*. In $\mathcal{S}$, this transition leads to *Readabc*. In $\mathcal{TP}_{comp}$, it leads to *S1*. Hence, there is a transition in the product, which starts at $\langle Start, Start \rangle$ and leads to $\langle Readabc, S1 \rangle$. This transition is labelled by the action *Init*. Its guard is *true* and its set of assignments is empty.
  - Now, $\mathcal{S}$ is in location *Readabc*. $\mathcal{TP}_{comp}$ is in location *S1*. Both IOSTS can fire one transition labelled by the same action: *Read*. In $\mathcal{S}$, this transition leads to *CheckPositive*. In $\mathcal{TP}_{comp}$, it loops back to *S1*. Hence, a new transition is constructed in $\mathcal{P}$. It is leading from $\langle Readabc, S1 \rangle$ to $\langle CheckPositive, S1 \rangle$. It is labelled by the common input action $Read?(p, q, r)$. The messages p, q, and r are the same as in the transition in $\mathcal{S}$. The guard of the newly created transition is the conjunction of the guards of the two transitions in $\mathcal{S}$ and $\mathcal{TP}_{comp}$, where each message of $\mathcal{TP}_{comp}$ is replaced by the corresponding message in $\mathcal{S}$. In this case, the new guard is *true*. The assignments carried by the new transition in $\mathcal{P}$ is the union of the two sets of assignments in $\mathcal{S}$ and $\mathcal{TP}_{comp}$, which is $\{a := p, b := q, c := r\} \cup \emptyset = \{a := p, b := q, c := r\}$.
  - The rest of the transitions in $\mathcal{P}$ is computed analogously to the above examples.

3.1.3 Closure

Test cases are intended to react promptly to inputs from the implementation under test. According to Rusu et al. [67], a straightforward way to ensure this, is to require input-completeness of the test case. In some cases, internal actions are hiding input actions. In order to generate test cases which are input-complete, all internal actions have to be removed from the product. This elimination of internal actions

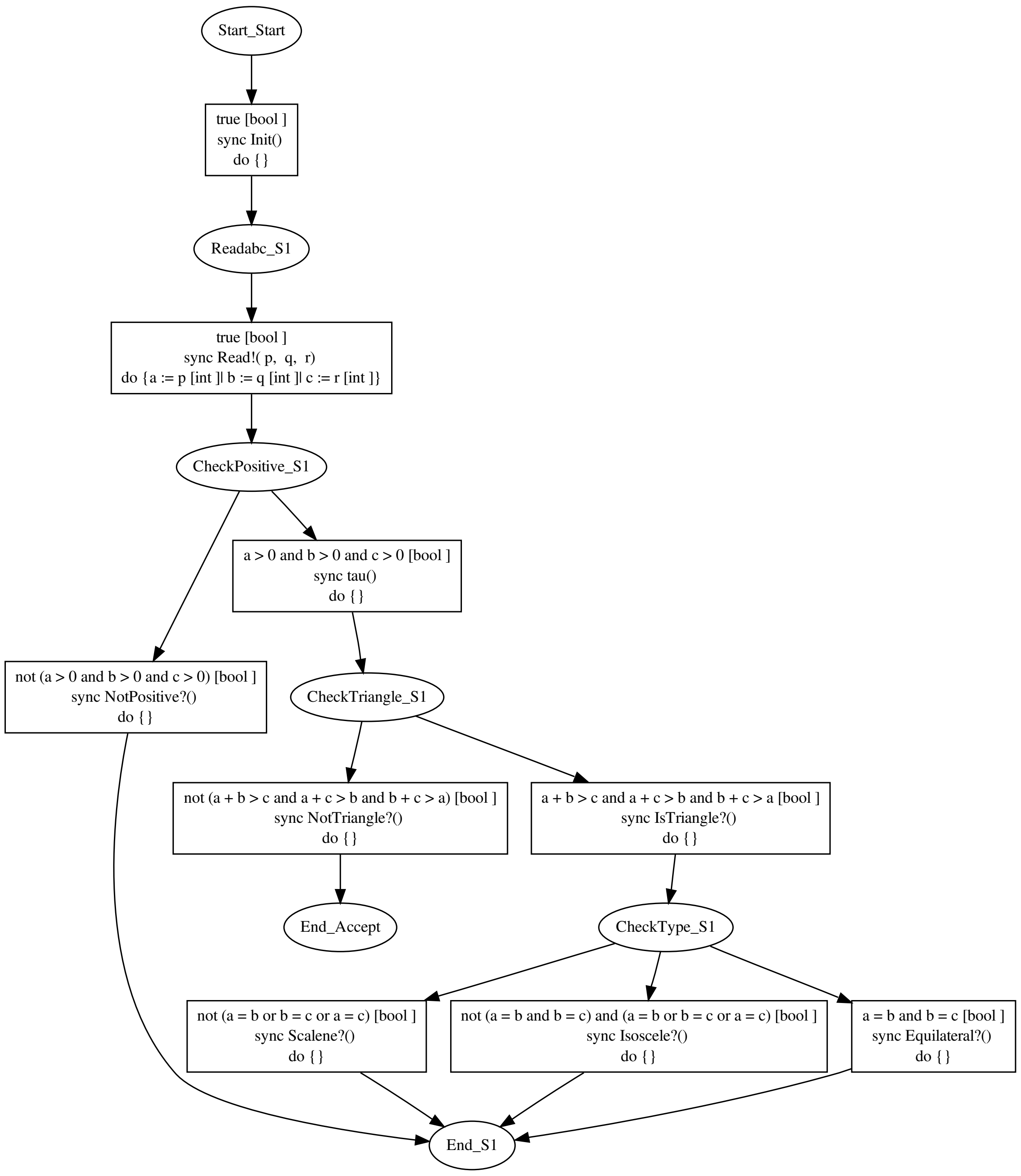
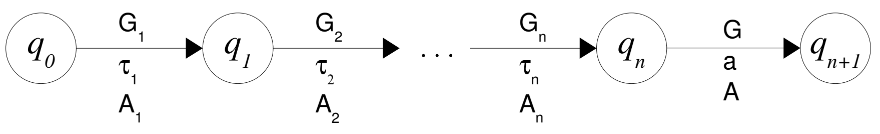


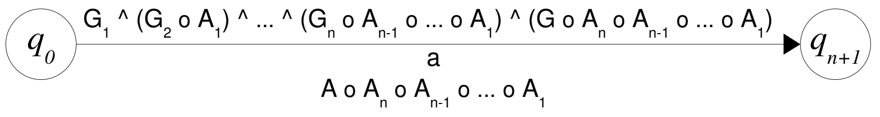
Figure 3.6: The IOSTS generated by STG when calculating the product of the IOSTS depicted in Figure 2.2 and the IOSTS of Figure 2.3.

is called closure and works for IOSTS without cycles of internal actions as depicted in Figure 3.7 and described below:

As depicted in Figure 3.7a, a sequence of internal actions $\tau_1, \tau_2, ..., \tau_n$, which is leading to an input action a, is given. The guard belonging to τ_i is G_i, the corresponding set of assignments is A_i for $1 \leq i \leq n$. The input action is guarded by G and triggers the assignments A. μ are the messages of action a. The whole sequence of symbolic transitions can be replaced by one symbolic transition $t = \langle q_0, a, \mu, G_1 \wedge (G_2 \circ A_1) \wedge ... \wedge (G_n \circ A_{n-1} \circ ... \circ A_1) \wedge (G \circ A_n \circ A_{n-1} \circ ... \circ A_1), A \circ A_n \circ A_{n-1} \circ ... \circ A_1, q_{n+1} \rangle$ as shown in Figure 3.7b. $f \circ g$ denotes composition of the functions f and g.



(a) Each sequence of internal actions leading to an input action is replaced by ...


q_0
$G_1 \wedge (G_2 \circ A_1) \wedge ... \wedge (G_n \circ A_{n-1} \circ ... \circ A_1) \wedge (G \circ A_n \circ A_{n-1} \circ ... \circ A_1)$
a
$A \circ A_n \circ A_{n-1} \circ ... \circ A_1$
q_{n+1}

(b) ... a sequence without internal actions.

Figure 3.7: The principle of the closure operation to eliminate internal actions.

An approach for dealing with cycles of internal actions for specifications with deterministic control structures like iterations and recursions was also proposed by Rusu et al. [67]. It is also indicated that the application of their closure algorithm results in an IOSTS with the same traces as the original one, but without internal actions. During experimentation with STG, it has been found out that the tool is not always able to eliminate all internal actions.

Example: Triangle Type Checker

Figure 3.8 depicts the IOSTS generated by STG after applying the closure operation on the product IOSTS of Figure 3.6. The product IOSTS contains one transition labelled with an internal action (from *CheckPositive_S1* to *CheckTriangle_S1*). This transition leads to two different input actions: *NotTriangle* labels the transition from *CheckTriangle_S1* to *End_Accept*. *IsTriangle* is the action of the transition from *CheckTriangle_S1* to *CheckType_S1*. These two transition sequences

1. $\textit{CheckPositive\_S1} \xrightarrow{G_1,\, \tau,\, A_1} \textit{CheckTriangle\_S1} \xrightarrow{G,\, a,\, A} \textit{End\_Accept}$
2. $\textit{CheckPositive\_S1} \xrightarrow{G_1,\, \tau,\, A_1} \textit{CheckTriangle\_S1} \xrightarrow{G',\, a',\, A'} \textit{CheckType\_S1}$

have to be closed. According to the above rules, each of the two transition sequences becomes one transition:

1. $\textit{CheckPositive\_S1} \xrightarrow{G_1 \wedge (G \circ A_1),\, a,\, A \circ A_1} \textit{End\_Accept}$
2. $\textit{CheckPositive\_S1} \xrightarrow{G_1 \wedge (G' \circ A_1),\, a',\, A' \circ A_1} \textit{CheckType\_S1}$

G_1 denotes the guard of the transition labelled with τ, which is $a > 0 \wedge b > 0 \wedge c > 0$. A_1 is the corresponding set of assignments, which is empty. G is the guard of the transition from *CheckTriangle_S1* to *End_Accept*, which is $\neg(a + b > c \wedge a + c > b \wedge b + c > a)$. a denotes the action of this transition, which is $NotTriangle?()$. The set of assignments for this transition is represented by A, which is the empty set again. Hence, the first transition sequence becomes one transition from *CheckPositive_S1* to *End_Accept* with

- guard $G_1 \wedge (G \circ A_1)$, which is $a > 0 \wedge b > 0 \wedge c > 0 \wedge \neg(a + b > c \wedge a + c > b \wedge b + c > a)$,
- action a, which is $NotTriangle?()$ and
- the set of assignments $A \circ A_1$, which is the empty set.

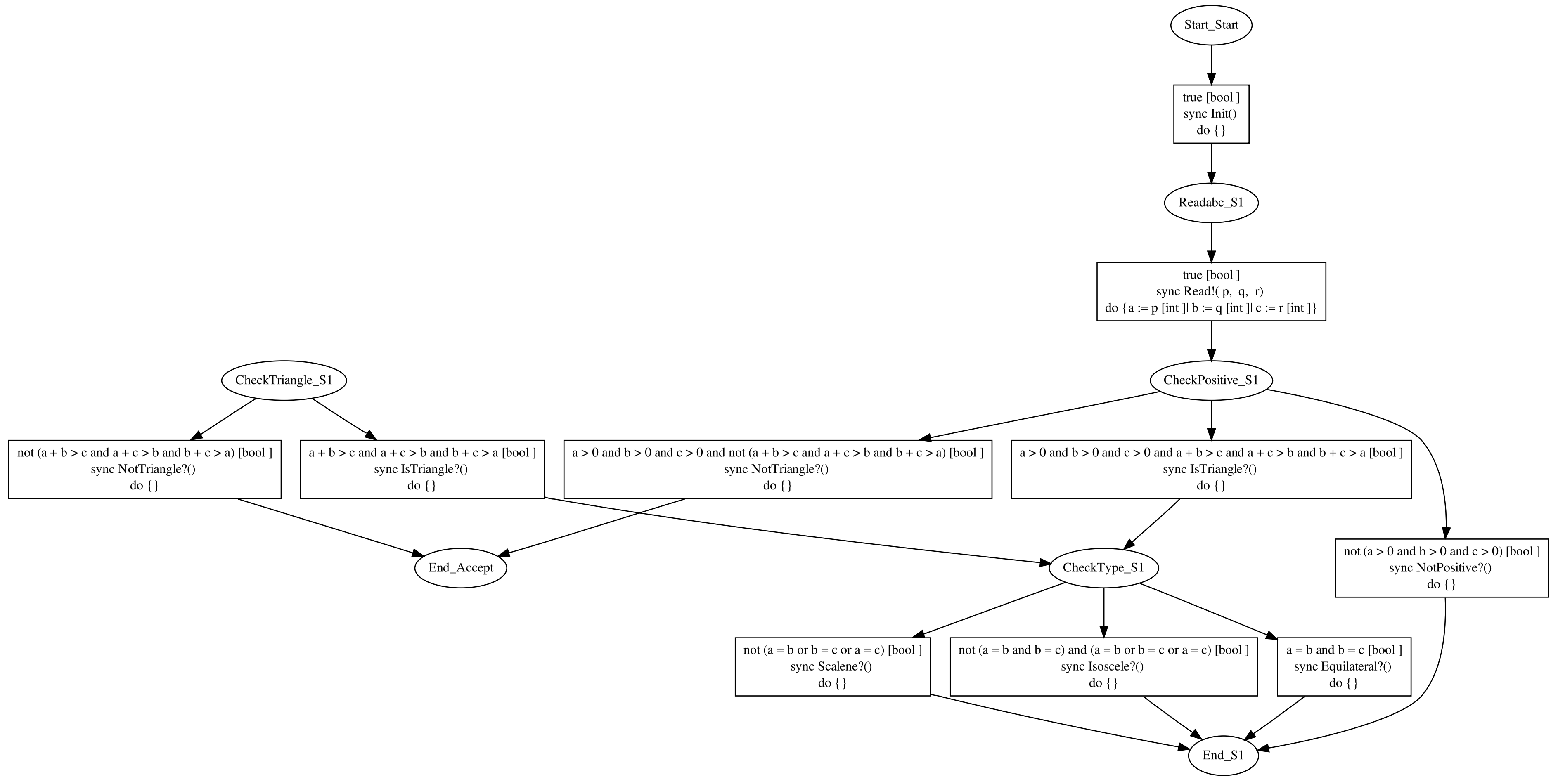

Start_Start
true [bool]
sync Init()
do { }
Readabc_S1
true [bool]
sync Read!(p, q, r)
do {a := p [int]| b := q [int]| c := r [int]}
CheckPositive_S1
a > 0 and b > 0 and c > 0 and a + b > c and a + c > b and b + c > a [bool]
sync IsTriangle?()
do { }
a > 0 and b > 0 and c > 0 and not (a + b > c and a + c > b and b + c > a) [bool]
sync NotTriangle?()
do { }
not (a > 0 and b > 0 and c > 0) [bool]
sync NotPositive?()
do { }
CheckTriangle_S1
not (a + b > c and a + c > b and b + c > a) [bool]
sync NotTriangle?()
do { }
a + b > c and a + c > b and b + c > a [bool]
sync IsTriangle?()
do { }
End_Accept
CheckType_S1
not (a = b or b = c or a = c) [bool]
sync Scalene?()
do { }
not (a = b and b = c) and (a = b or b = c or a = c) [bool]
sync Isoscele?()
do { }
a = b and b = c [bool]
sync Equilateral?()
do { }
End_S1

Figure 3.8: The product IOSTS (see Figure 3.6) after closure.

G' denotes the guard of the transition from *CheckTriangle_S1* to *CheckType_S1*, which is $a + b > c \wedge a + c > b \wedge b + c > a$. a' is the action labelling this transition, which is $IsTriangle?()$. A' represents the set of assignments for this transition, which is the empty set. Hence, the second transition sequence becomes one transition from *CheckPositive_S1* to *CheckType_S1* with

- guard $G_1 \wedge (G' \circ A_1)$, which is $a > 0 \wedge b > 0 \wedge c > 0 \wedge a + b > c \wedge a + c > b \wedge b + c > a$,
- action a', which is $IsTriangle?()$ and
- the set of assignments $A' \circ A_1$, which is the empty set.

STG removes the internal transition from the IOSTS and adds the two new transitions as described above. STG does not remove the middle location *CheckTriangle_S1*, because other transitions may lead to it. The outgoing transitions of this middle location are kept as well. Hence, the IOSTS still has the location *CheckTriangle_S1*, which has no incoming transitions any more, but still has its outgoing transitions labelled with *NotTriangle* and *IsTriangle*.

3.1.4 Determinization

In order to avoid a dependency of the verdicts on the internal choices of the tester, test cases must be deterministic. The goal of the determinization step is to compute an IOSTS which has no non-deterministic choices and is trace-equivalent to the closed product of $\mathcal{S}$ and $\mathcal{TP}$ [67].

Figure 3.9 is based on a figure of Rusu et al. [67] and shows a non-deterministic IOSTS and the trace-equivalent IOSTS which is computed via determinization. A typical case of a non-deterministic choice is depicted in Figure 3.9a. The same action a leads to two different locations and/or causes different context updates. Since determinizing symbolic transition systems is difficult in general, Rusu et al. [67] use a heuristic, which deals with common situations like the one depicted in Figure 3.9a. The basic idea is to delay effects of internal actions until non-determinism can be resolved. Figure 3.9b shows the application of this idea on the IOSTS of Figure 3.9a: The two symbolic transition, which are both labelled with action a and therefore cause non-determinism, are split into three transitions guarded by:

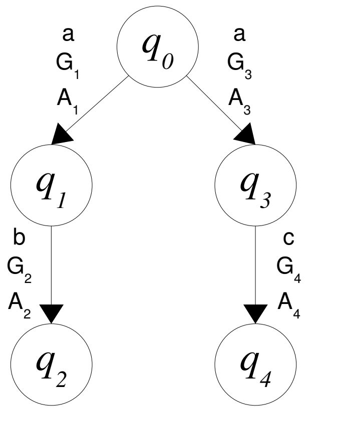
1. $G_1 \wedge \neg G_3$: It is clear that the next location will be q_1.
2. $G_3 \wedge \neg G_1$: It is clear that the next location will be q_3.
3. $G_1 \wedge G_3$: In this case, the next location can either be q_1 or $q3$. The assignments are postponed until the next observable action. A new location $q_{1/3}$ is added. If b is the next action, the assignments A_1 should have been carried out. If the next action is c, A_3 should have been used. The execution of the delayed assignments is accomplished by composing A_1 (resp. A_3) with (a) the guard G_2 (resp. G_4) and (b) the assignments of the observable action A_2 (resp. A_4).

Of course, if b and c were the same action, the whole procedure would have to be applied again and it might not terminate. According to Zinovieva-Leroux [80], the current version of STG does not support determinization yet.

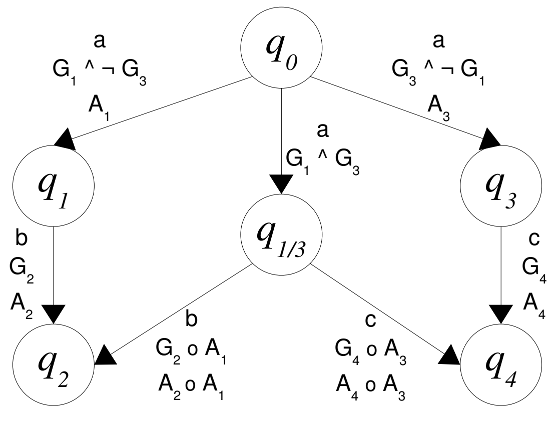
3.1.5 Test Case Selection: Reachability and Coreachability

According to Zinovieva-Leroux [80], the IOSTS which has been generated so far (see Figure 3.8) is already a valid test case. Nevertheless, it is bigger than necessary. Some locations may be unreachable, e.g., in Figure 3.8, it is obvious that the location *CheckTriangle_S1* cannot be reached. For simplification and selection of a smaller but still correct test case, reachability and coreachability analyses are performed.

The selection algorithm relies on the sets of reachable and coreachable states of an IOSTS. The construction of these sets is not directly implemented in STG. They are calculated by the verification and



(a) A non-deterministic IOSTS: Starting from q_0, the same action a leads to two different locations and/or causes different context updates.



(b) IOSTS after determinization: A heuristic is used, which delays effects of internal actions until the next observable action is reached. Therefor, a new location has to be introduced.

Figure 3.9: The principle of the determinization operation.

slicing tool NBac (see Section 3.3.1). In general, reachability and coreachability problems are undecidable, i.e., an IOSTS may have an unbounded state space. Hence, NBac calculates over-approximations of the sets of reachable and coreachable states [80].

In the following, it will be explained how the sets of reachable and coreachable states are defined, but not how their approximations are calculated. Furthermore, the selection algorithm used in the STG tool will be explained.

Set of Reachable States The set of reachable states contains all states s of an IOSTS $\mathcal{I}$, for which the following condition holds: $\exists\sigma \in traces(\mathcal{I}),\ \exists s_0 \in S_0 : s_0 \overset{\sigma}{\Rightarrow} s$. In other words, a state s of an IOSTS $\mathcal{I}$ belongs to the set of reachable states if there exists a trace σ that ends in s [80].

Set of Coreachable States The set of coreachable states contains all states s of an IOSTS $\mathcal{I}$, for which the following condition holds: $\exists\sigma \in (\Upsilon \cup \Omega)^*,\ \exists s_{acc} \in S_{acc} : s \overset{\sigma}{\Rightarrow} s_{acc}$ whereby S_{acc} denotes the set of *Accept* states. In other words, a state s of an IOSTS $\mathcal{I}$ belongs to the set of coreachable states if it is possible to go from s to some accepting state s_{acc} [80].

Test Case Selection Algorithm

The selection algorithm is intended to generate an IOSTS with fewer unreachable states, which may lead to the satisfaction of the used test purpose [80]. According to Zinovieva-Leroux [80], the inversion of input and output actions is also performed during test case selection. Actually, STG has already inverted inputs to outputs and vice versa during the calculation of the product (see Section 3.1.2).

The selection algorithm performed by STG consists of three steps[3]:

1. First Reachability Analysis:
   STG calculates the set of reachable states of the closed product IOSTS with the help of NBac. All unreachable states and not fireable transitions (transitions with an unsatisfiable guard) will be removed from the IOSTS [80]. The resulting IOSTS will be called $\mathcal{I}_{reach}$ in the following.

[3]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_2.html` (last visit 2009-09-27)

2. Coreachability Analysis:
   STG calculates the set of coreachable states of the IOSTS resulting from the first reachability analysis ($\mathcal{I}_{reach}$) with the help of NBac. When the set of coreachable states is known, the following steps are performed on $\mathcal{I}_{reach}$ to generate the resulting IOSTS $\mathcal{I}_{coreach}$ [80]:

   (a) All transitions labelled by an output action and leaving the set of coreachable states will be removed. If such a transition was executed, it would not be possible to reach an *Accept* state any more. Since output means output of the test case, the transition can be removed to keep a chance of reaching an *Accept* state.

   (b) All transitions labelled by an input action and leaving the set of coreachable states will be redirected to the location *Inconclusive*. Since input means input from the implementation under test, the transition can not be removed. Each valid input must be accepted by the test case. The transition will be redirected to an *Inconclusive* location, because after executing this transition no *Accept* state can be reached any more.

   (c) The guards of the other transitions will be modified by taking into account the information about coreachable states. This information includes conditions on the IOSTS data (system parameters, variables, and messages), which correspond to weakest preconditions (introduced by Dijkstra [31]). These conditions can be used to strengthen the guards of other transitions in order to increase the probability to reach some *Accept* state and to avoid inconclusiveness.

       This guard strengthening may require to add transitions starting at the same location as the strengthened transition and leading to the location *Inconclusive*. This is particularly the case when the guard of a transition labelled by an input action is strengthened. In this case, it is necessary to ensure that the test case still rejects only non-conformant implementations. The guard of this new transition is a conjunction of the original guard and the negation of the strengthening condition.

3. Second Reachability Analysis:
   Since the coreachability analysis could have produced unreachable states in the resulting IOSTS $\mathcal{I}_{coreach}$, a second reachability analysis is performed.

Example: Triangle Type Checker

The Triangle Type Checker example will be used to explain STG's test case selection algorithm by looking at the results of the reachability/coreachability analyses.

First Reachability Analysis: Figure 3.10 depicts STG's result of the first reachability analysis, which has been applied to the closed product IOSTS shown in Figure 3.8. It is clear that the location *CheckTriangle_S1* in Figure 3.8 is not in the set of reachable states, since (a) it is not the initial location of the IOSTS and (b) it has no predecessors. During the reachability analysis, all unreachable locations will be removed. Hence, *CheckTriangle_S1* and its transitions

- *CheckTriangle_S1* $\rightarrow$ *End_Accept*
- *CheckTriangle_S1* $\rightarrow$ *CheckType_S1*

will not be part of the IOSTS $\mathcal{I}_{reach}$ any more. The other locations are reachable and part of $\mathcal{I}_{reach}$.

The guards of the transitions have been transformed by NBac into BDDs (Binary Decision Diagrams), respectively MTBDDs (Multi-Terminal Binary Decision Diagrams), by applying the Shannon Cofactor Expansion. Since an elaboration on (MT)BDDs and Shannon Cofactor Expansion is not essential for understanding STG's test generation process, we refer to corresponding literature (e.g., Fujita et al. [39]) for further information about this topic.

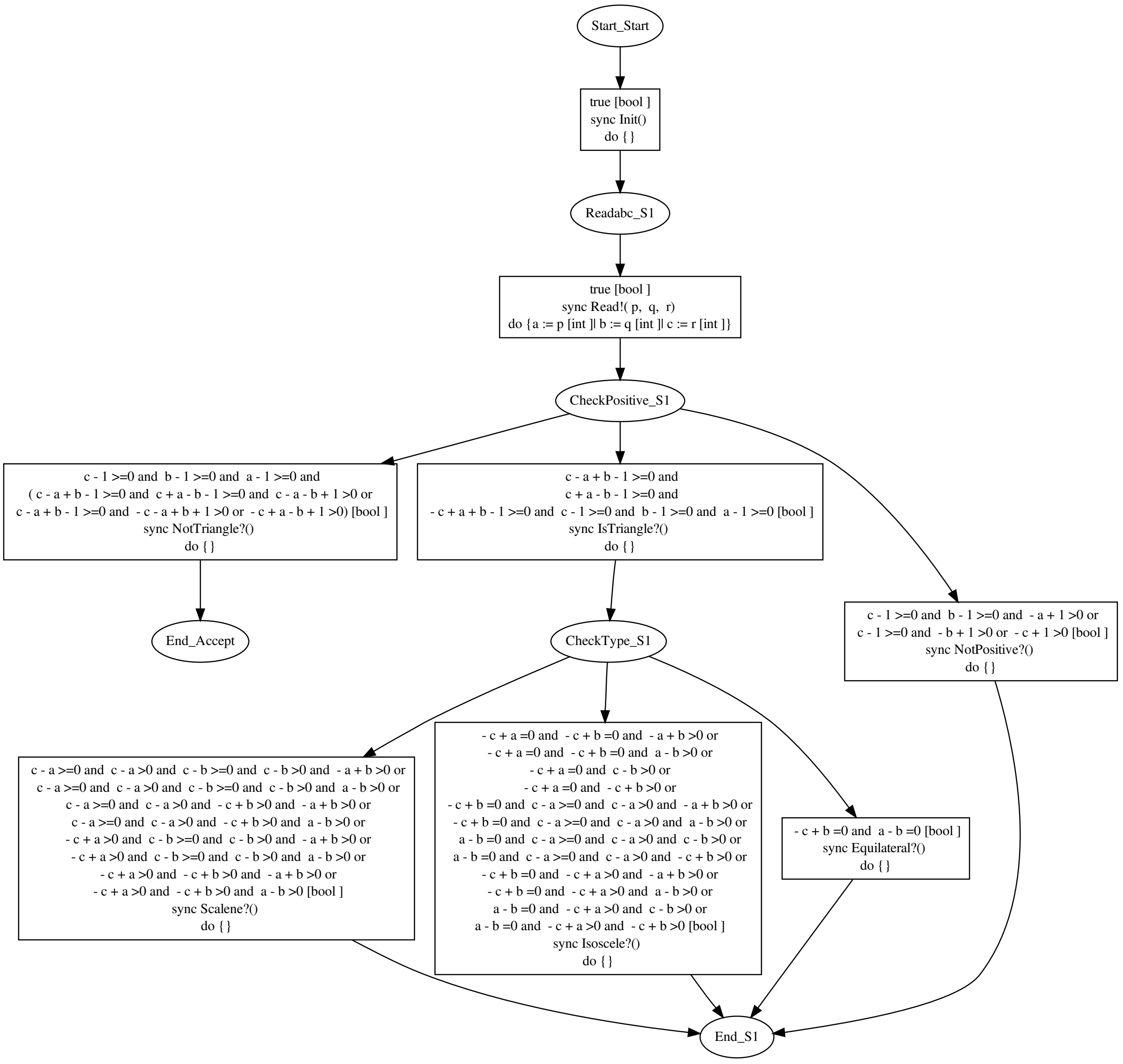

Start_Start
true [bool]
sync Init()
do { }
Readabc_S1
true [bool]
sync Read!(p, q, r)
do {a := p [int]| b := q [int]| c := r [int]}
CheckPositive_S1
c - 1 >=0 and b - 1 >=0 and a - 1 >=0 and
(c - a + b - 1 >=0 and c + a - b - 1 >=0 and c - a - b + 1 >0 or
c - a + b - 1 >=0 and - c - a + b + 1 >0 or - c + a - b + 1 >0) [bool]
sync NotTriangle?()
do { }
c - a + b - 1 >=0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and c - 1 >=0 and b - 1 >=0 and a - 1 >=0 [bool]
sync IsTriangle?()
do { }
c - 1 >=0 and b - 1 >=0 and - a + 1 >0 or
c - 1 >=0 and - b + 1 >0 or - c + 1 >0 [bool]
sync NotPositive?()
do { }
End_Accept
CheckType_S1
c - a >=0 and c - a >0 and c - b >=0 and c - b >0 and - a + b >0 or
c - a >=0 and c - a >0 and c - b >=0 and c - b >0 and a - b >0 or
c - a >=0 and c - a >0 and - c + b >0 and - a + b >0 or
c - a >=0 and c - a >0 and - c + b >0 and a - b >0 or
- c + a >0 and c - b >=0 and c - b >0 and - a + b >0 or
- c + a >0 and c - b >=0 and c - b >0 and a - b >0 or
- c + a >0 and - c + b >0 and - a + b >0 or
- c + a >0 and - c + b >0 and a - b >0 [bool]
sync Scalene?()
do { }
- c + a =0 and - c + b =0 and - a + b >0 or
- c + a =0 and - c + b =0 and a - b >0 or
- c + a =0 and c - b >0 or
- c + a =0 and - c + b >0 or
- c + b =0 and c - a >=0 and c - a >0 and - a + b >0 or
- c + b =0 and c - a >=0 and c - a >0 and a - b >0 or
a - b =0 and c - a >=0 and c - a >0 and c - b >0 or
a - b =0 and c - a >=0 and c - a >0 and - c + b >0 or
- c + b =0 and - c + a >0 and - a + b >0 or
- c + b =0 and - c + a >0 and a - b >0 or
a - b =0 and - c + a >0 and c - b >0 or
a - b =0 and - c + a >0 and - c + b >0 [bool]
sync Isoscele?()
do { }
- c + b =0 and a - b =0 [bool]
sync Equilateral?()
do { }
End_S1

Figure 3.10: The IOSTS $\mathcal{I}_{reach}$ generated by STG after the first reachability analysis applied to the IOSTS depicted in Figure 3.8.

Coreachability Analysis: Figure 3.11 shows the IOSTS $\mathcal{I}_{coreach}$ generated by STG during the coreachability analysis. The set of coreachable states has been calculated for the IOSTS $\mathcal{I}_{reach}$ shown in Figure 3.10. The states with the locations *End_Accept*, *CheckPositive_S1*, *Readabc_S1*, and *Start_Start* are coreachable. When the set of coreachable states is known, the following actions can be performed to generate $\mathcal{I}_{coreach}$ out of $\mathcal{I}_{reach}$:

- The transitions leaving *CheckPositive_S1*, leading to a not coreachable location, and labelled by an input action will be redirected to a newly introduced location *inconc* (inconclusive). The affected transitions and their replacements are:

$$\textit{CheckPositive\_S1} \xrightarrow{IsTriangle?} \textit{CheckType\_S1} \quad \Rightarrow \quad \textit{CheckPositive\_S1} \xrightarrow{IsTriangle?} \textit{inconc}$$

$$\textit{CheckPositive\_S1} \xrightarrow{NotPositive?} \textit{End\_S1} \quad \Rightarrow \quad \textit{CheckPositive\_S1} \xrightarrow{NotPositive?} \textit{inconc}$$

- Due to the redirection of the transition leading from *CheckPositive_S1* to *CheckType_S1*, the latter does not have any incoming transitions any more. Since *CheckType_S1* is not coreachable, its outgoing transitions will be redirected to the location *inconc* as well.

- The guard of the transition from *Readabc_S1* to *CheckPositive_S1* has been strengthened by adding the condition $p - 1 \geq 0 \wedge q - 1 \geq 0 \wedge r - 1 \geq 0$, which is equivalent to $p > 0 \wedge q > 0 \wedge r > 0$ for integer values. Since this transition is labelled by an output action, no additional transition leading to an inconclusive location is necessary.
- NBac tries to strengthen the guard of the transition from *CheckPositive_S1* to *End_Accept*, although no new information can be added. Thus, the guard looks more complicated, but is equivalent to the guard of the same transition in $\mathcal{I}_{reach}$ from Figure 3.10. Since this transition is labelled by an input action and it has been tried to strengthen its guard, a transition from *CheckPositive_S1* to *inconc* is introduced. Its guard is the conjunction of the original guard and the negation of the strengthening condition. Since the original guard could not be strengthened, the guard of the newly added transition is the same as the original one. This means that the guard of the new transition is not satisfiable.

Second Reachability Analysis: Figure 2.4 has already shown the final test case, whereby the *Fail* locations are still missing. This IOSTS is generated by applying the second reachability analysis on the IOSTS depicted in Figure 3.11.

CheckType_S1 is unreachable. Hence, it has been removed together with its outgoing transitions. The transition labelled with *NotTriangle* from *CheckPositive_S1* to *inconc* has been removed, because its guard is not satisfiable. Note that this transition has been added due to guard strengthening during coreachability analysis.

The transition from *CheckPositive_S1* to *inconc* labelled by *IsTriangle* is kept in the resulting IOSTS. Its guard can be made weaker, because the guard of the only preceding transition from *Readabc_S1* to *CheckPositive_S1* has been strengthened by $p - 1 \geq 0 \wedge q - 1 \geq 0 \wedge r - 1 \geq 0$. Since the assignments of this transition are $a := p \mid b := q \mid c := r$, the condition $c - 1 \geq 0 \wedge b - 1 \geq 0 \wedge a - 1 \geq 0$ is assured in this state of the IOSTS and it can be removed from the guard.

For the same reason, the guard of the transition from *CheckPositive_S1* to *inconc* labelled with *NotPositive* has become unsatisfiable. The conjunction of the original guard $(c-1 \geq 0 \wedge b-1 \geq 0 \wedge -a+1 > 0 \vee c - 1 \geq 0 \wedge -b + 1 > 0 \vee -c + 1 > 0)$ and $c - 1 \geq 0 \wedge b - 1 \geq 0 \wedge a - 1 \geq 0$ is not satisfiable. Hence, this transition has to be removed.

3.1.6 Input-Completion and Transformation into Java

As defined in Section 2.3.4, a test case has to be input-complete except for the locations *Accept*, *Reject*, and *Inconclusive*. This is not yet the case for the generated test case of Figure 2.4. Zinovieva-Leroux [80] presents the following algorithm for making a generated test case $\mathcal{TC}$ input-complete:

For each location $q \in Q_{\mathcal{TC}} \setminus \{$*Accept, Inconclusive, Reject*$\}$ and each input action $a \in \Sigma^{i}_{\mathcal{TC}}$:

1. If there are one or more transitions starting at q which are labelled with input action a, and if the conjunction of the guards of these transitions is not equal to *true*, then a new transition originating at q has to be created. It leads to the new location *Fail* (respectively *Reject*), is labelled by a, and is guarded by the negated disjunction of the guards of the other transitions labelled with a and starting from q.
2. If there is no transition labelled with a and starting at q, a new transition starting at q and leading to *Fail/Reject* has to be created. It is labelled with a and its guard is *true*.

In both cases, the set of assignments of the newly introduced transitions is the so-called set of identity assignments. This means that each variable of $\mathcal{TC}$ is assigned to itself.

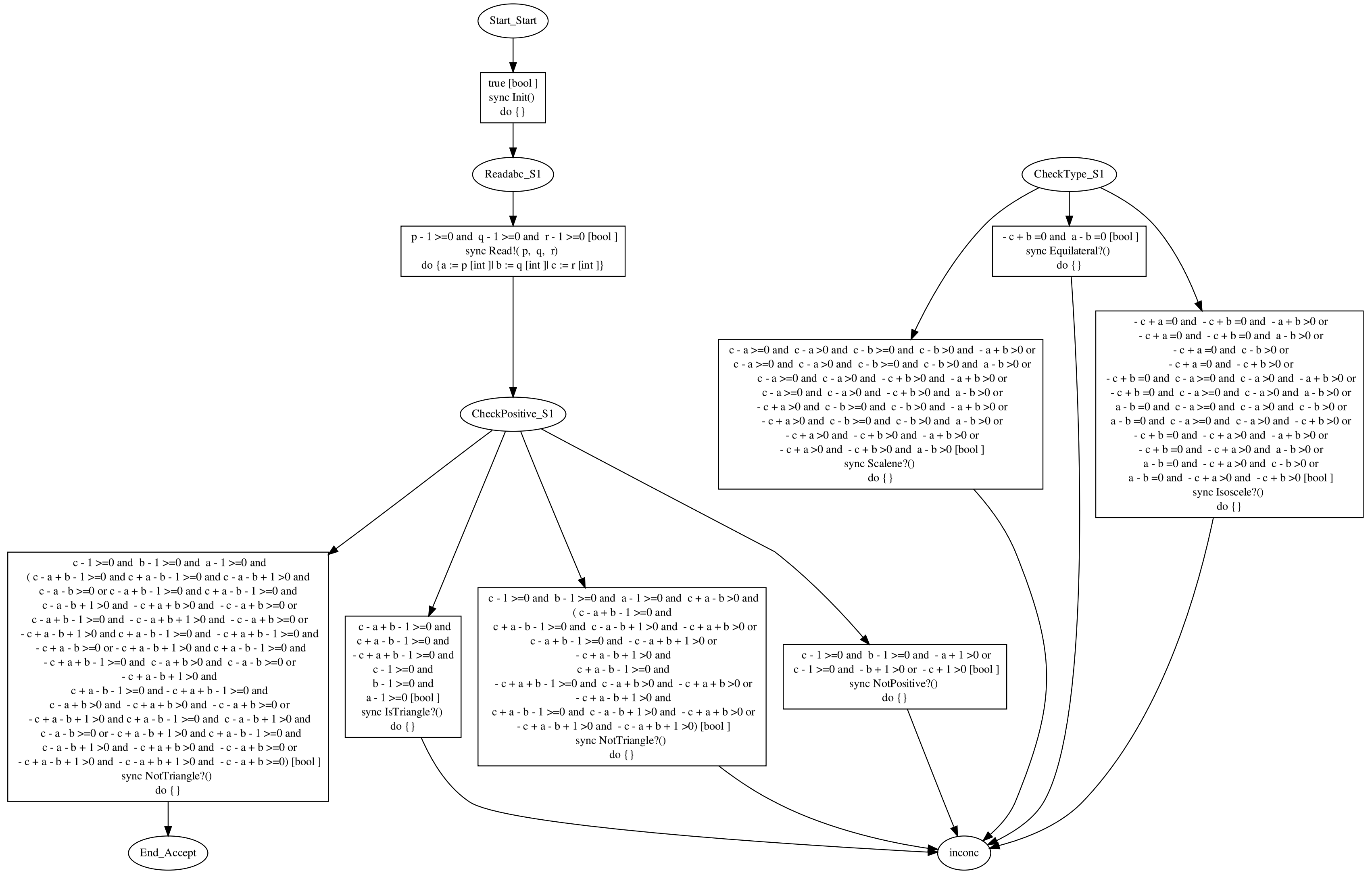

Start_Start
true [bool]
sync Init()
do { }
Readabc_S1
p - 1 >=0 and q - 1 >=0 and r - 1 >=0 [bool]
sync Read!(p, q, r)
do {a := p [int]| b := q [int]| c := r [int]}
CheckPositive_S1
c - 1 >=0 and b - 1 >=0 and a - 1 >=0 and
(c - a + b - 1 >=0 and c + a - b - 1 >=0 and c - a - b + 1 >0 and
c - a - b >=0 or c - a + b - 1 >=0 and c + a - b - 1 >=0 and
c - a - b + 1 >0 and - c + a + b >0 and - c - a + b >=0 or
c - a + b - 1 >=0 and - c - a + b + 1 >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and c + a - b - 1 >=0 and - c + a + b - 1 >=0 and
- c + a - b >=0 or - c + a - b + 1 >0 and c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and c - a + b >0 and c - a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and - c + a + b - 1 >=0 and
c - a + b >0 and - c + a + b >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and c + a - b - 1 >=0 and c - a - b + 1 >0 and
c - a - b >=0 or - c + a - b + 1 >0 and c + a - b - 1 >=0 and
c - a - b + 1 >0 and - c + a + b >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and - c - a + b + 1 >0 and - c - a + b >=0) [bool]
sync NotTriangle?()
do { }
End_Accept
c - a + b - 1 >=0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and
c - 1 >=0 and
b - 1 >=0 and
a - 1 >=0 [bool]
sync IsTriangle?()
do { }
c - 1 >=0 and b - 1 >=0 and a - 1 >=0 and c + a - b >0 and
(c - a + b - 1 >=0 and
c + a - b - 1 >=0 and c - a - b + 1 >0 and - c + a + b >0 or
c - a + b - 1 >=0 and - c - a + b + 1 >0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and c - a + b >0 and - c + a + b >0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and c - a - b + 1 >0 and - c + a + b >0 or
- c + a - b + 1 >0 and - c - a + b + 1 >0) [bool]
sync NotTriangle?()
do { }
c - 1 >=0 and b - 1 >=0 and - a + 1 >0 or
c - 1 >=0 and - b + 1 >0 or - c + 1 >0 [bool]
sync NotPositive?()
do { }
inconc
CheckType_S1
- c + b =0 and a - b =0 [bool]
sync Equilateral?()
do { }
c - a >=0 and c - a >0 and c - b >=0 and c - b >0 and - a + b >0 or
c - a >=0 and c - a >0 and c - b >=0 and c - b >0 and a - b >0 or
c - a >=0 and c - a >0 and - c + b >0 and - a + b >0 or
c - a >=0 and c - a >0 and - c + b >0 and a - b >0 or
- c + a >0 and c - b >=0 and c - b >0 and - a + b >0 or
- c + a >0 and c - b >=0 and c - b >0 and a - b >0 or
- c + a >0 and - c + b >0 and - a + b >0 or
- c + a >0 and - c + b >0 and a - b >0 [bool]
sync Scalene?()
do { }
- c + a =0 and - c + b =0 and - a + b >0 or
- c + a =0 and - c + b =0 and a - b >0 or
- c + a =0 and c - b >0 or
- c + a =0 and - c + b >0 or
- c + b =0 and c - a >=0 and c - a >0 and - a + b >0 or
- c + b =0 and c - a >=0 and c - a >0 and a - b >0 or
a - b =0 and c - a >=0 and c - a >0 and c - b >0 or
a - b =0 and c - a >=0 and c - a >0 and - c + b >0 or
- c + b =0 and - c + a >0 and - a + b >0 or
- c + b =0 and - c + a >0 and a - b >0 or
a - b =0 and - c + a >0 and c - b >0 or
a - b =0 and - c + a >0 and - c + b >0 [bool]
sync Isoscele?()
do { }

Figure 3.11: The IOSTS $\mathcal{I}_{coreach}$ generated by STG after the coreachability analysis applied to the IOSTS depicted in Figure 3.10.

The process of making the test case input-complete is implemented within the transformation of the IOSTS test case into a Java (respectively SJava, see Section 3.2.1) test case. The generated Java test case is ready to be executed against an implementation under test. The test case execution process will be elaborated in the next section of this chapter.

Example: Triangle Type Checker

Figure 3.12 shows the final test case. It was generated by making the test case resulting from the second reachability analysis (see Figure 2.4) input-complete. Four transitions leading to the location *Fail* have been added. To keep the IOSTS graph well-arranged, some transitions have been merged.

The transition from *Readabc_S1* to *Fail* represents six transitions. Each of them has the same source and destination location. They only differ in their actions. The set $\{*\}$ denotes the six input actions *NotPositive*, *NotTriangle*, *IsTriangle*, *Equilateral*, *Isoscele*, and *Scalene*. These input actions are labelling the six merged transitions. The transition from *CheckPositive_S1* to *Fail* labelled by $\{*\} \setminus \{$*NotTriangle*, *IsTriangle*$\}$ represents four transitions labelled by the input actions *NotPositive*, *Equilateral*, *Isoscele*, and *Scalene*. These transitions were created according to Rule 2 of the algorithm for making a test case input-complete.

Rule 1 of the input-completion algorithm causes the other two transitions which have been added. Since there are two transitions labelled with an input action (*NotTriangle* and *IsTriangle*) originating from *CheckPositive_S1* in Figure 2.4, two transitions labelled with the same input actions leading to the *Fail* location have been added in Figure 3.12. Each of these transitions is guarded by the negated guard of the corresponding existing transition. The captions of the transitions labelled with *NotTriangle* have been coloured red. The captions of the transitions labelled with *IsTriangle* are blue. The guards of the transitions leading to *Fail* have been abbreviated for clarity.

3.2 Test Case Execution with STG

This section is intended to explain how STG executes the generated test cases. STG uses external components for test case execution, which will be introduced in the beginning of this section. Afterwards, requirements for interfacing with the implementation under test will be discussed. Finally, the test execution process will be explicated.

3.2.1 External Tools

For the execution of the generated test case, two external tools are employed by STG: the SJava compiler and the constraint solver Lucky. Both tools are included in the STG backend package available in the download section of the STG web page[4].

SJava

SJava[5], shorthand for Synchronous Java, was developed at EPFL (École Polytechnique fédérale de Lausanne), Switzerland. It is an extension of the Java programming language designed to simplify concurrency issues in Java. Although SJava source code has to be compiled with its own compiler, the produced byte code is executable with any Java Runtime Environment.

SJava is based on the concept of Synchronous Active Objects (SAO), which was introduced by Petitpierre [59]. In an active object, a thread is tightly bound to the object. Each active object has a `run` method, which defines its behaviour. The active object's thread is started at the instantiation of the object

[4]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)

[5]`http://ltiwww.epfl.ch/sJava/` (last visit 2009-09-27)

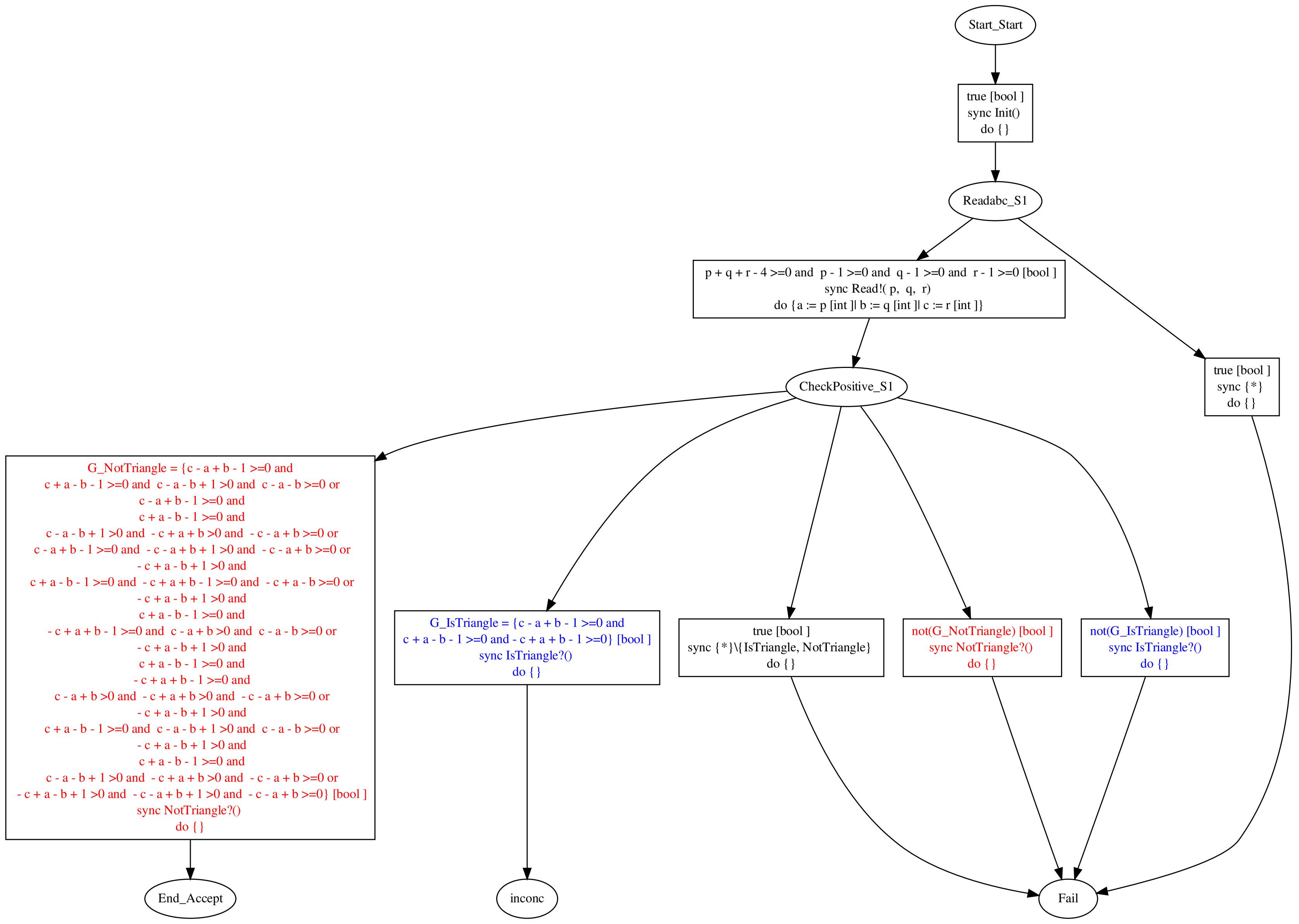
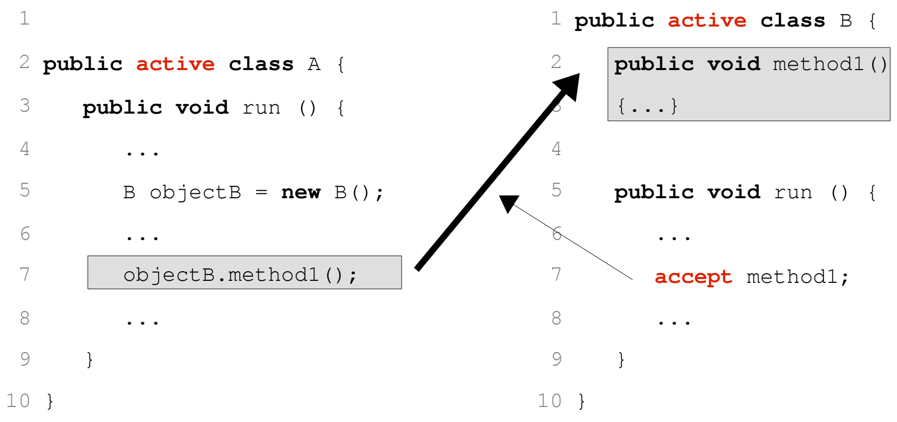
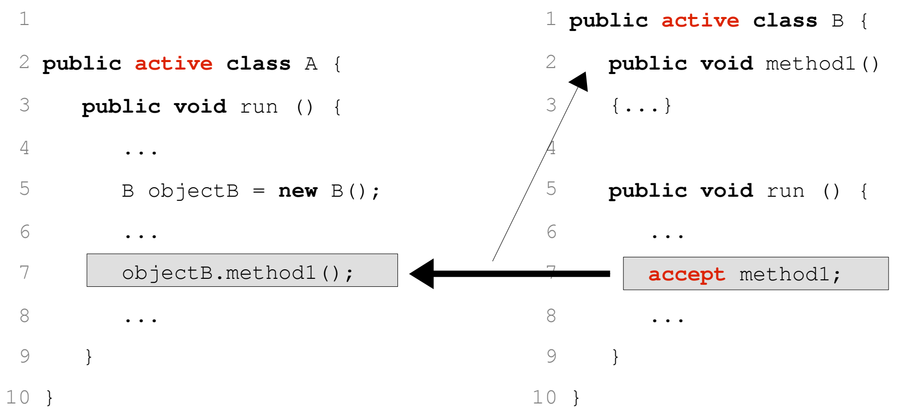

Start_Start
true [bool]
sync Init()
do { }
Readabc_S1
p + q + r - 4 >=0 and p - 1 >=0 and q - 1 >=0 and r - 1 >=0 [bool]
sync Read!(p, q, r)
do {a := p [int]| b := q [int]| c := r [int]}
true [bool]
sync {*}
do { }
CheckPositive_S1
G_NotTriangle = {c - a + b - 1 >=0 and
c + a - b - 1 >=0 and c - a - b + 1 >0 and c - a - b >=0 or
c - a + b - 1 >=0 and
c + a - b - 1 >=0 and
c - a - b + 1 >0 and - c + a + b >0 and - c - a + b >=0 or
c - a + b - 1 >=0 and - c - a + b + 1 >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and - c + a + b - 1 >=0 and - c + a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and c - a + b >0 and c - a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
- c + a + b - 1 >=0 and
c - a + b >0 and - c + a + b >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and c - a - b + 1 >0 and c - a - b >=0 or
- c + a - b + 1 >0 and
c + a - b - 1 >=0 and
c - a - b + 1 >0 and - c + a + b >0 and - c - a + b >=0 or
- c + a - b + 1 >0 and - c - a + b + 1 >0 and - c - a + b >=0} [bool]
sync NotTriangle?()
do { }
G_IsTriangle = {c - a + b - 1 >=0 and
c + a - b - 1 >=0 and - c + a + b - 1 >=0} [bool]
sync IsTriangle?()
do { }
true [bool]
sync {*}\{IsTriangle, NotTriangle}
do { }
not(G_NotTriangle) [bool]
sync NotTriangle?()
do { }
not(G_IsTriangle) [bool]
sync IsTriangle?()
do { }
End_Accept
inconc
Fail

Figure 3.12: The final test case produced by making the IOSTS of Figure 2.4 input-complete.



```
 1                                      1 public active class B {
 2 public active class A {              2     public void method1()
 3     public void run () {             3     {...}
 4         ...                          4
 5         B objectB = new B();         5     public void run () {
 6         ...                          6         ...
 7         objectB.method1();           7         accept method1;
 8         ...                          8         ...
 9     }                                9     }
10 }                                   10 }
```

(a) Perspective of the caller (objectA): objectA calls objectB's method1 and blocks until objectB accepts this call.



```
 1                                      1 public active class B {
 2 public active class A {              2     public void method1()
 3     public void run () {             3     {...}
 4         ...                          4
 5         B objectB = new B();         5     public void run () {
 6         ...                          6         ...
 7         objectB.method1();           7         accept method1;
 8         ...                          8         ...
 9     }                                9     }
10 }                                   10 }
```

(b) Perspective of the callee (objectB): When objectB gets to the accept statement in line 7, it blocks until its method1 has been executed.

Figure 3.13: SJava's synchronization mechanism can be seen as a rendezvous between two Synchronous Active Objects.

and killed when the object is destructed. In SJava programs, every thread has to be encapsulated into an object. This object's class is marked with the keyword `active`.

Two active objects communicate via method calls, which have to be synchronized because of concurrency. This synchronization is implemented via the keyword `accept`. Figure 3.13 is an adaption of a figure created by Petitpierre [59] and shows the two perspectives of this synchronization. Figure 3.13a depicts the caller's point of view (view of *objectA*). *objectA* calls *objectB*'s `method1` and blocks until *objectB* accepts this call. The callee's perspective, the view of *objectB*, is shown in Figure 3.13b. When *objectB* gets to the `accept` statement in line 7, *objectB* blocks until its `method1` has been executed. Therefore, the synchronization can be seen as a rendezvous (as defined by Hoare [45]) between two SAOs in which both objects/threads have to be in line 7 (call statement, respectively accept statement). During their rendezvous, the called method is executed.

For more sophisticated applications, `select`, `when` (guard) and `waituntil` (timeout) statements have been introduced in SJava. Listing 3.1 shows an example of a `select` statement according to Petitpierre [59]. The first statement in each `case` has to be a so-called trigger: a method invocation, a possibly guarded `accept` statement, or a timeout. `case` statements with guards evaluating to $false$ are omitted for the current execution of the `select` statement.

```
public void run () {
   ...
   for (;;) {
      select {
         case
            result = object.methodA(argument);
            ...
         case
            when (guard) accept methodB;
            ...
         case
            waituntil (currentTimeMillis()+1000);
            ...
      }
   }
}
```

Listing 3.1: Example of the SJava `select` statement.

Lucky

For test case execution, STG employs a constraint solver. According to Jeannet et al. [49], this constraint solver is used to perform the following two tasks:

1. It is used to generate concrete values for the messages of test case outputs, such that the corresponding guard is fulfilled and the goal specified in the used test purpose can be achieved.
2. It is used to decide which transition can be fired when the test case receives some input from the implementation.

For the STG execution framework, the constraint solver Lucky[6] was selected to perform these tasks.

3.2.2 Implementation under Test

In order to execute test cases with the STG backend, an interface for the implementation under test (IUT) has to be implemented. In the literature, an exact specification of this interface is hard to find. Two rather outdated sources (Belinfante et al. [10] and Clarke et al. [24]) provide basic information about how this interface should look like. They deal with an earlier version of STG, which generated C++ test cases.

According to Clarke et al. [24], the IUT is assumed to be a C++ class. The IUT and the test case are executed in parallel during testing. They communicate via method invocations: inputs to the IUT become synchronous method calls with arguments representing the action's messages. The output from the IUT is the return value of the method invocation. In other words, the interface class for the IUT has to implement a method for each action of the test case. These methods must have the same signatures as their corresponding actions, so that the action's messages can be passed through the method's parameters [10].

As already mentioned, the current version of STG generates Java test cases and literature on the IUT's interface concerning the current version of STG is hard to find. In the following, the Triangle Type Checker example will be used to demonstrate how the interface for the IUT is supposed to look like in order to work with the current version of STG.

[6]`http://www-verimag.imag.fr/~synchron/lurette/lucky.html` (last visit 2009-09-27)

Example: Triangle Type Checker

`triangle.java` contains the implementation of the Triangle Type Checker example and is included in the archive `stg_backend.tgz` (folder `EssaiTri`). The archive can be downloaded from the STG web page[7]. The class *triangle* implements the Java *Runnable* interface, since the implementation and the test case will be executed in parallel during testing. It uses different classes provided by the STG backend. They are located in the sub-folder `fichiersjavaainclure` and implement the Java *Serializable* interface:

- *IfGate* represents the gates/actions of an IOSTS.
- *IostsMess* represents an IOSTS action and its messages.
- *MessInt* is derived from *IostsMess* and represents actions with integer messages.
- *Void* represents the empty message.

The implementation breaks with the principles introduced by Belinfante et al. [10] and Clarke et al. [24]. The communication of the implementation and the tester is not realized by direct method invocation, but via streams. There is an *OutputStream* for the implementation's output and an *InputStream* for the input received by the implementation. The methods provided by the implementation are:

- *void run()* implements the main program logic.
- *void setInput(InputStream)* sets the *InputStream*.
- *void setOutput(OutputStream)* sets the *OutputStream*.
- *void send(IostsMess)* sends the specified IOSTS action with its messages by writing it to the *OutputStream*.
- *void NotPositive()* is called if the output action *NotPositive* should be sent. It constructs the corresponding *IostsMess* and sends it by invoking the *send* method. For each action (*NotTriangle*, *Scalene*, ...), one method is implemented like the one for *NotPositive*.

The class providing the interface to the implementation must have the same name as the specification file. For example, if the system specification is in the file `triangle.stg`, the interface to the implementation has to be in the file `triangle.java`.

3.2.3 Test Case Execution Process

In the following, a description of how STG executes its generated test cases will be given. The main roles and their interactions will be described. For better understanding, a sequence diagram for the Triangle Type Checker example will be presented.

The Actors and their Interactions

The main actors in the execution process of STG's test cases are:

1. The implementation under test.
2. The tester, which is the test case generated by STG.
3. The implementation manager, which is also created by STG during test case generation.

[7]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)

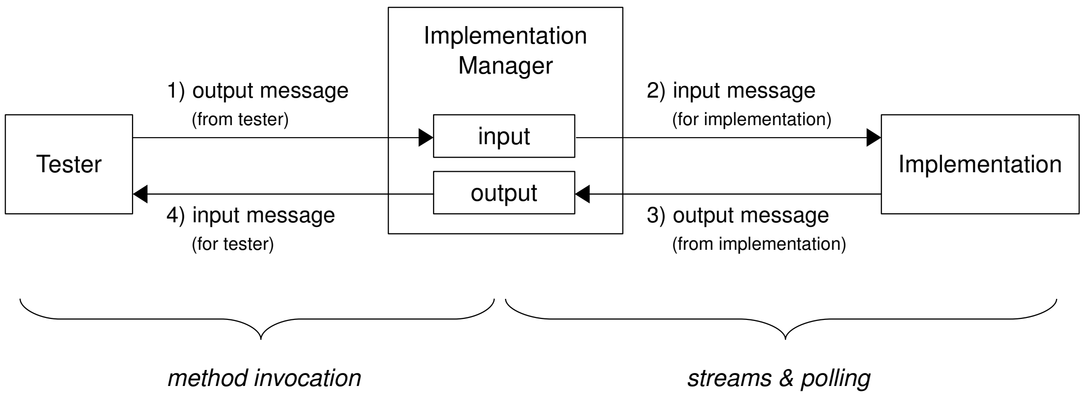

Implementation Manager
input
output
Tester
Implementation
1) output message (from tester)
2) input message (for implementation)
3) output message (from implementation)
4) input message (for tester)
method invocation
streams & polling

Figure 3.14: A schematic illustration of STG's test case execution: The tester's output is not sent directly to the implementation under test (IUT), but to the implementation manager. The latter is a mediator between the tester and the IUT and forwards the received output to the IUT, which receives it as input. The same applies for communication into the other direction.

These three actors and their relationships are represented in Figure 3.14. The tester sends some output, which is intended for the implementation under test (IUT). This output is not sent directly to the IUT, but to the implementation manager, which is a mediator between the tester and the IUT. The implementation manager forwards the received output from the tester to the IUT, which receives it as input. The same applies for communication into the other direction.

The mechanism used for communication between the implementation manager and the IUT differs from the one used for communication between the implementation manager and the tester. The tester and the implementation manager interact via method invocations, while the IUT and the implementation manager interact via streams and polling.

Three files, which have been generated during test case generation (see Section 3.1), will be used during test execution:

TestDrive.sjava is the main class for test execution. If the implementation under test has parameters, it parses them from the command line. Additionally, it reads in a user-defined delay value, which will be used as timeout value during message passing. The test driver instantiates the implementation interface class, the test class, and the implementation manager. The implementation manager gets three important pieces of information from the test driver: the tester (an instance of the test class), the implementation under test (an instance of the implementation interface class) and the delay value, which was read in before.

ImpManager.sjava contains the implementation manager class. The most important methods provided by this class are:

- *void receive(IostsMess)* is used for receiving messages from the tester. It forwards the input from the tester to the implementation under test. This method is called by the tester to send its output to the implementation via the implementation manager.
- *void checkOutput()* checks if some output from the implementation is available (see Figure 3.14, "output"). If output from the implementation is available, it can be forwarded to the tester by invoking the tester's *receive* method.
- *void run()* sets the implementation manager's input and output streams needed for communication with the implementation under test. It starts the implementation under test in a new thread and subsequently executes a loop until the test case has finished.
  The loop contains an SJava `select` statement (see Section 3.2.1), which distinguishes three

cases: The first one applies if some output from the implementation has to be forwarded to the tester. Then the tester's *receive* method is called.
The second and the third case apply if there is nothing to forward. The second case accepts a call of the implementation manager's own *receive* method and starts a timeout. The value for the timeout (*delay*) has been specified by the user when starting the test execution. The third case calls the tester's *release* method. Hence, if the implementation manager's *receive* method is not invoked while it is accepted, the implementation manager calls the tester's *release* method.
After each execution of the `select` statement, the decision parameters are updated. In other words, it is checked if the implementation has sent a message.

The implementation manager and the tester communicate by invoking each others' *receive* methods. For interaction with the IUT, the implementation manager employs streams and polling.

basenameTest.sjava is the SJava class for the generated test case. *basename* stands for the name of the STG specification file. For example, for the specification file `triangle.stg`, the test class file will be named `triangleTest.sjava`.
The test class uses a symbolic transition system for modelling the generated test case. The base classes of this transition system can be found in the folder `fichiersjavaainclure` of the archive `stg_backend.tgz`, which can be downloaded from the STG web page[8]. The test class extends these base classes by adding input and output transitions to the locations (class *IostsState*) and by adding actions, guards, and assignments to the transitions (class *IostsTrans*). It calls the Lucky constraint solver to generate concrete input values for the implementation under test and for validating the guard to determine if a transition is fireable or not.

The most important methods provided by this class are:

- *void receive(IostsMess)* is used for receiving messages from the implementation manager. This method is called by the implementation manager to forward output from the implementation under test to the tester. *receive* additionally causes the transition system to execute a fireable transition of the current location and switch over to the next location.
- *void release()* causes the test case to fire a transition and move on to the next location of the transition system.
- *void run()* initializes the transition system and executes a loop until no more transition in the current state of the transition system is ready to be fired. The loop contains an SJava `select` statement (see Section 3.2.1), which has three cases:
  The first one applies if a message is ready to be sent to the implementation manager (the implementation under test respectively). The *receive* method of the implementation manager will be called in order to deliver the message to the implementation under test. The second and the third case statement apply if no message is ready for sending. Then either a call of the tester's own *receive* (case 2) or of its *release* (case 3) method is accepted.
  In other words, if there is a message ready to be sent, it will be transmitted to the implementation manager. Otherwise the tester is either expecting to receive a message and to fire a transition of the transition system or expecting a command to fire a transition without receiving a message.

Example: Triangle Type Checker

For a better understanding of the roles and their interactions, which were explained previously, the sequence diagram of an example execution shall be discussed.

Figure 3.15 is the sequence diagram for an execution of the test case depicted in Figure 2.4. At first, the test driver instantiates the class interfacing the implementation. In the Triangle Type Checker

[8]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)

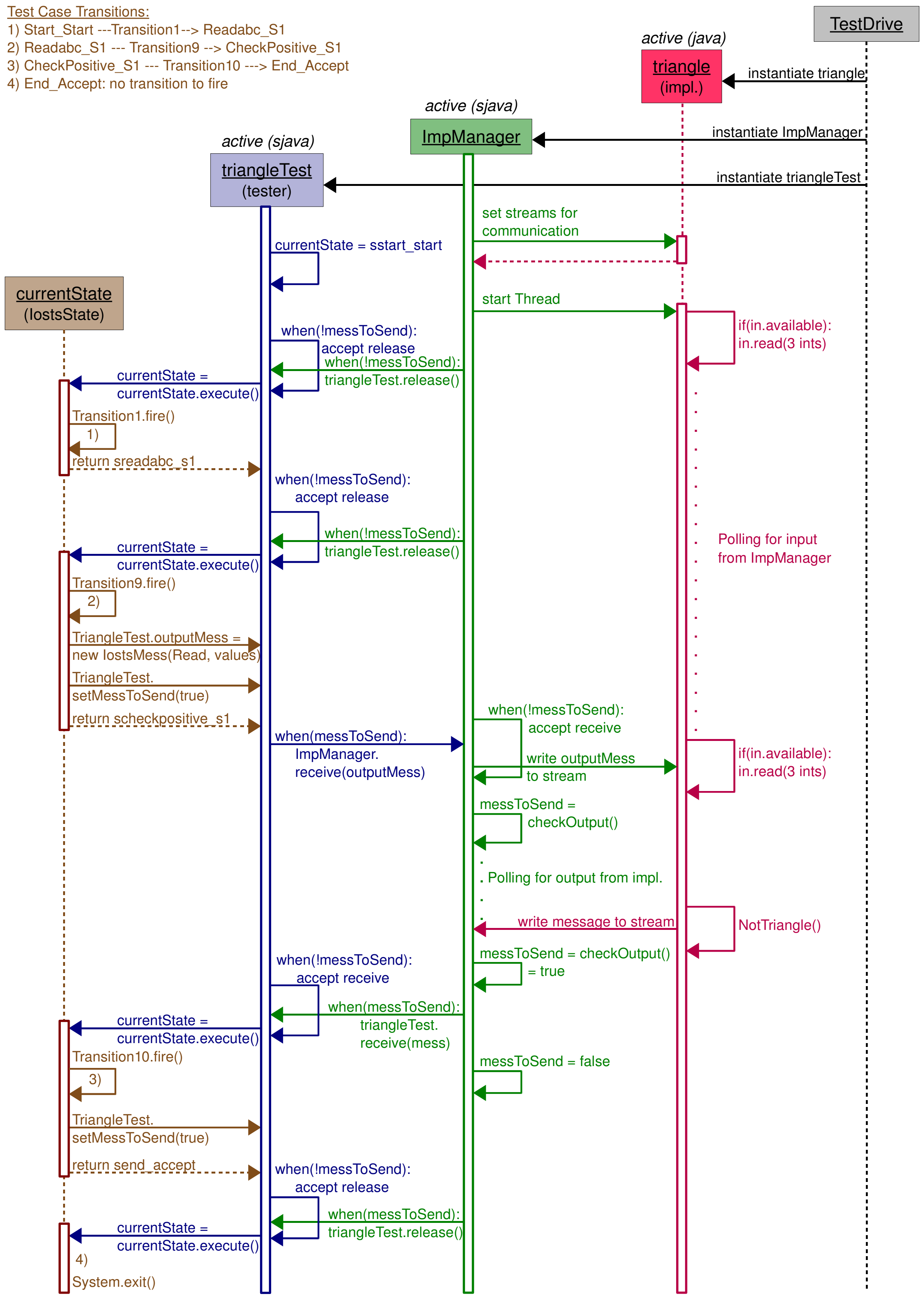

Test Case Transitions:
1) Start_Start ---Transition1--> Readabc_S1
2) Readabc_S1 --- Transition9 --> CheckPositive_S1
3) CheckPositive_S1 --- Transition10 ---> End_Accept
4) End_Accept: no transition to fire
TestDrive
active (java)
triangle (impl.)
instantiate triangle
active (sjava)
ImpManager
instantiate ImpManager
active (sjava)
triangleTest (tester)
instantiate triangleTest
set streams for communication
currentState = sstart_start
currentState (lostsState)
start Thread
if(in.available): in.read(3 ints)
when(!messToSend): accept release
when(!messToSend): triangleTest.release()
currentState = currentState.execute()
Transition1.fire()
1)
return sreadabc_s1
when(!messToSend): accept release
when(!messToSend): triangleTest.release()
Polling for input from ImpManager
currentState = currentState.execute()
Transition9.fire()
2)
TriangleTest.outputMess = new lostsMess(Read, values)
TriangleTest. setMessToSend(true)
return scheckpositive_s1
when(!messToSend): accept receive
when(messToSend): ImpManager. receive(outputMess)
write outputMess to stream
if(in.available): in.read(3 ints)
messToSend = checkOutput()
Polling for output from impl.
write message to stream
NotTriangle()
when(!messToSend): accept receive
messToSend = checkOutput() = true
when(messToSend): triangleTest. receive(mess)
currentState = currentState.execute()
Transition10.fire()
3)
messToSend = false
TriangleTest. setMessToSend(true)
return send_accept
when(!messToSend): accept release
when(messToSend): triangleTest.release()
currentState = currentState.execute()
4)
System.exit()

Figure 3.15: Sequence diagram for a test execution of the test case shown in Figure 2.4.

example, this class is called *triangle* and is depicted in red in the diagram. Subsequently, the test driver instantiates the implementation manager (green) and the test class (blue), which is called *triangleTest* in this example. Since the implementation manager and the tester are declared *active* (see Section 3.2.1 about SJava), instantiating them also means invoking their *run* methods.

By now, two threads have been started: The tester initializes its transition system by setting the current state and waits for input from the implementation manager. In the meantime, the implementation manager connects itself with the implementation by setting the streams needed for communication and starts the implementation in a new thread. Henceforward, each of the main actors has its own thread. The tester, the implementation, and the implementation manager run in parallel and have to synchronize each other. The tester and the implementation manager handle synchronization via mechanisms like `accept` and `select` statements introduced by SJava (see Section 3.2.1). The implementation and the implementation manager communicate via polling for input data from the other party.

The implementation under test expects to receive three integer values as input and has no output to send. The tester has nothing to send in its current state as well and accepts either a call of its *receive* or of its *release* method. The implementation manager detects that no input will be received - neither from the tester nor from the implementation. In this case, it calls the tester's *release* method and causes the test case to move on in its transition system. The tester fires *Transition1* and traverses from state "Start_Start" to state "Readabc_S1".

This scenario repeats once more, since still no input is ready to be sent: The tester moves from state "Readabc_S1" to state "CheckPositive_S1" by firing *Transition9*. By executing this transition, the tester produces input data for the implementation and sends the message to the implementation manager by invoking its *receive* method. The implementation manager accepts this method call and forwards the input from the tester to the implementation under test by writing the message to the implementation's input stream. The implementation, which is still polling for input, detects the message and processes it. The result "NotTriangle" will be written to the implementation's output stream.

The implementation manager, which is polling for output from the implementation, detects the message and forwards it to the tester by invoking its *receive* method. The tester accepts the call and executes *Transition10*, which leads from state "CheckPositive_S1" to state "End_Accept" and whose action is "NotTriangle". Once again, the implementation manager calls the tester's *release* method. The tester is now in state "End_Accept", which has no more outgoing transitions. Since no more transition can be fired, the test execution is aborted. The resulting verdict is "Pass".

3.3 STG Usage

3.3.1 Installation

This section is supposed to give installation support for the STG tool. As there are many pieces of software which have to be installed and configured properly in order to run STG, this document is supposed to be an installation help by reporting already made experiences.

As a prerequisite for a successful installation of the STG tool, the following programs and libraries have to be installed and configured properly:

1. NBac
2. MLCuddIDL
3. Lucky
4. Graphviz -Dotty

The required installation packages can be found in the download section of the official STG web page[9].

[9]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)

Generally, installation instructions are provided by each software package, which should contain sufficient information to install the respective tool. In the following, additional hints will be given and experienced problems will be described.

NBac

NBac is a verification and slicing tool developed by Bertrand Jeannet. According to the official NBac web page[10], reachability (forward) and coreachability (backward) analyses and a combination thereof can be performed. STG employs NBac to execute reachability and coreachability analyses to prune the test case [49]. It is not required to explicitly install NBac, since the required binaries are included in the STG packages.

MLCuddIDL

According to Bertrand Jeannet's MLCuddIDL web page[11], MLCuddIDL is a C library allowing to interface with the CUDD BDD library for OCaml. The following requirements need to be met for a proper installation of MLCuddIDL:

1. gcc (version 4.2.3)
2. CUDD BDD library (version 2.4.1)
3. CUDDAUX library (not versioned)
4. OCaml (version 3.10.0)
5. CamlIDL stub code generator (version 1.05)

The parenthesized numbers state the version numbers used in the reference installation of this thesis.

It is recommended to keep the installation order from the list above, because some of the tools and libraries are built on their predecessors in the list. For example, CUDDAUX needs the CUDD BDD library to be already installed.

A notably remark is that each of the above tools except of OCaml was downloaded and installed manually according to the installation instructions in the corresponding README file. OCaml was installed via `apt-get install ocaml` on a Debian-based system.

For the installation of MLCuddIDL, ocamlc.opt is required. According to the manpages of ocamlc, ocamlc.opt is not available in all installations of Objective Caml. This was the case in the reference installation of this work. Therefore a new folder `bin` had to be created in the ocaml installation directory (e.g., `/usr/lib/ocaml/3.10.0/bin`) and the following two symbolic links had to be added in this new `bin` directory as well as in `/usr/bin`:

1. *ocamlc.opt* links to the file *ocamlc*, which is located in `/usr/bin` for the reference installation.
2. *ocamlopt.opt* links to the file *ocamlopt*, which is located in `/usr/bin` for the reference installation.

This workaround can be used without risk, because according to the ocamlc manpages, *ocamlc.opt* behaves exactly the same as *ocamlc* but it compiles faster. The same holds for *ocamlopt.opt* and *ocamlopt*.

[10] `http://pop-art.inrialpes.fr/people/bjeannet/nbac/index_1.html` (last visit 2009-09-27)

[11] `http://mlxxxidl.gforge.inria.fr/mlcuddidl/index.html` (last visit 2009-09-27)

Lucky

Lucky[12] is a constraint solver used in the STG execution framework to produce appropriate test case inputs and to check constraints. The STG execution framework includes an already compiled library of LUCKY, which makes an additional installation unnecessary.

Graphviz - Dotty

Graphviz is a program widely used for graph visualization [34]. It can either be downloaded and installed by following the download link on the Graphviz web page[13] or preferably by using the command `apt-get install graphviz` on Debian-based systems.

STG

After the installation of all required tools, STG can be installed. There are two packages available for download[14], which are both needed:

1. `stg_linux.tgz`: The test generation tool itself, which offers to generate Java test cases.
2. `stg_backend.tgz`: The test execution framework, which allows to execute the generated tests against a given implementation.

Both packages need to be extracted. The best way to run both components of STG is to use shell scripts, which will be introduced in Section 3.3.2.

Note that STG is not versioned. The files used for this installation have been downloaded from the STG web page[15] dated with *August 28, 2008*.

3.3.2 Executing STG

This section gives a short description of how to automatically generate and execute test cases with STG.

Test Case Generation

As already mentioned in Section 3.3.1, the test generation component of STG is contained in the archive `stg_linux.tgz`. To generate Java test cases, the following two IOSTS are needed:

1. A specification of the system.
2. A test purpose to select the part of the specification which shall be tested.

The most comfortable way to generate test cases with STG is to use shell scripts. Listing 3.2 shows the script we used for test case generation. It can be reused with adoption of the user-specific paths.

At first, the directory `stg_linux`, which includes the STG executable file, has to be added to the `PATH` variable. In Line 5, the *rung* file is copied from `stg_linux/tmp` to the global `tmp` directory as described in the INSTALL file of `stg_linux`. Since the global `tmp` directory is purged from time to time, the most convenient way to ensure the existence of the *rung* file is to automatically copy it every time.

The script expects the STG specification file to be located in a directory called `spec` (see Line 8 in Listing 3.2). After changing into the `spec` directory, the required NBac library files, which are located in

[12]`http://www-verimag.imag.fr/~synchron/lurette/lucky.html` (last visit 2009-09-27)
[13]`http://www.graphviz.org/` (last visit 2009-09-27)
[14]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)
[15]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)

```
 1 # extend the PATH variable by adding the path to the stg_linux folder
 2 export PATH=$PATH:/stg_linux
 3
 4 # copy "rung" file in order to be able to execute the stg tool
 5 cp /stg_linux/tmp/rung /tmp/rung
 6
 7 # go to directory "spec" (contains specification file <name.stg>)
 8 cd spec
 9
10 # create folder "lib" in working directory and copy NBac library into it
11 mkdir -p ./lib/nbac
12 cp /stg_linux/lib/nbac/* ./lib/nbac
13
14 # call stg to generate "java test program", "ImpManager" and "TestDrive"
15 # from specification file with default operations
16 # stg <stg_spec_file.stg> -test_name <test> -test_purpose_name <TP>
17 stg triangle.stg -test_name Triangle -test_purpose_name TP1
18 # stg triangle.stg -cmdfile <cmdfile>
```

Listing 3.2: Shell script for test case generation with STG.

`stg_linux/lib`, are copied directly into the `spec` directory. This is necessary, because STG expects the NBac library to be in the same folder as the specification file it is processing.

Finally, STG is called. Line 17 shows the simplest way of invoking STG by specifying the input file, the process name of the specification, and the process name of the test purpose in this file. This command prompts STG to execute its default operation sequence (see Section 3.1 for an explanation of these operations):

1. completion (of the test purpose specification)
2. product (between the specification and the test purpose)
3. closure (of the product)
4. reach (first reachability analysis)
5. coreach (coreachability analysis)
6. reach (second reachability analysis)
7. tojava (transformation of the resulting test case into Java classes)

An alternative way of calling STG is shown as a comment in Line 18: the definition of the operation sequence via a command file. More detailed information about STG's program options and its command line syntax can be found in the "Reference Manual" section of the STG web page[16].

Test Case Execution

The package `stg_backend.tgz` represents the test execution component of STG (see Section 3.3.1). Listing 3.3 is an adapted and enhanced shell script of the one that can be found in section "Executing the test" of the STG web page[17].

At first, the script has to set the path to an already installed Java environment and some other paths to files in `stg_backend` (Lines 1 to 7). In the course of this thesis, Java SDK 1.4 was used because problems with higher versions of Java were encountered.

[16]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_4.html` (last visit 2009-09-27)
[17]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_11.html` (last visit 2009-09-27)

```
 1 # set java paths
 2 export JAVA_HOME=/java_sdk_1.4/j2sdk1.4.2_18
 3
 4 # set backend paths needed for execution
 5 export BACKEND=/stg_backend
 6 export CLASSPATH=$BACKEND/sjava/lib/sjavaEnv.jar:$BACKEND/sjava/bin/
       sjavac.jar:$BACKEND/solveur/:.
 7 export LD_LIBRARY_PATH=$BACKEND/solveur/
 8
 9 # create a directory to gather all is needed and go to it
10 mkdir exec
11 cd exec
12
13 # copy generic files (Enum, IfGate, IostsMess, IostsState, ...)
14 cp $BACKEND/fichiersjavaainclure/*.java .
15
16 # copy specific files: TestDrive.sjava, ImpManager.sjava and
17 #                      <test_file>.sjava
18 cp ../spec/*.java .
19 cp ../spec/*.sjava .
20
21 # copy the implementation program
22 cp ../impl/*.java .
23
24 # before compiling, remove old class-files
25 # (sjava does not seem to recompile automatically)
26 rm *.class
27
28 # compile everything with sjava
29 # ATTENTION: List Lire.java separately (otherwise Segmentation Fault)
30 $BACKEND/sjava/bin/sjavac -label *sjava Lire.java
31
32 # execute the test
33 java TestDrive
```

Listing 3.3: Shell script for test case execution with STG.

The best way to keep track of the files needed for test execution and the generated Java test class files is to put them altogether into an empty folder. Therefor, a directory named `exec` is created and set to be the new working directory (Line 10 and Line 11). Afterwards, the following Java classes (respectively SJava classes) have to be copied into this directory:

1. Generic files (Line 14):
   Every Java file located in the folder `fichiersjavaainclure` of the STG backend component. These files involve for example `IostsMess.java` and `IostsState.java` and are the same for each test case execution scenario.
2. Specific files (Line 18 and Line 19):
   Every SJava[18] and Java file generated by STG in the test generation phase (thus different for each test case execution scenario and located in the `spec` directory).
3. Implementation files (Line 22):
   The implementation of the SUT with all of its files. The script in Listing 3.3 expects the Java implementation files to be located in the directory `impl`.

[18] `http://ltiwww.epfl.ch/sJava/` (last visit 2009-09-27)

Test Purpose	# Transitions incl. Duplicates	# Duplicate Transitions	Redundancy
1 register_delete_tp	611	49	8.01 %
2 register_invalidrequest_tp	201	18	8.96 %
3 register_notfound_tp	201	18	8.96 %
4 register_ok_tp	239	22	9.21 %
5 register_unauthorized_tp	529	47	8.88 %
6 register_notfound_guarded_tp	211	0	0 %
Sum	1992	154	–
Average	332	26	7.83 %

Table 3.1: This table presents statistics about redundant transitions in the closed products from the SIP Registrar specification and five different test purposes.

The used SJava compiler does not seem to recompile already existing class files automatically, although there were changes. Thus, already existing class files are removed (Line 26) to ensure that all class files correspond to the latest version of their source files.

Line 30 shows the invocation of the SJava compiler `sjavac` (see Section 3.2.1), which is included in the STG backend package. There is one important remark for this line: The file `Lire.java` has to be passed as a separate argument to the SJava compiler. If not, a segmentation fault occurs and the compilation does not work. Finally, Line 33 starts the test execution by invoking the `TestDrive` class.

3.4 Encountered Problems

In the following, deficiencies that have been identified in the current implementation of STG will be discussed. The affected operations are: product calculation, closure, determinization, and test case selection (reachability and coreachability).

3.4.1 Duplicate Transitions in the Product

While using STG to generate test cases for the Session Initiation Protocol (SIP) Registrar (see Section 6.2) and the Conference Protocol (see Section 6.3), it has been observed that the product of the specification and the test purpose contains a large number of duplicate transitions. These transitions are redundant, since they are exactly the same. They have the same start and destination locations and their guards, actions, messages, and assignments are equal.

For example, the closed products generated for the SIP Registrar specification and six different test purposes include 26 redundant transitions on average, i.e., each closed product could be reduced by approximately 8 % of its transitions. The Conference Protocol results in even more considerable values. At an average, each closed product for the Conference Protocol specification and ten different test purposes contains 149 redundant transitions, i.e., the number of transitions per product could be reduced by about 30 %. Table 3.1 and Table 3.2 give detailed information about the exact values for each test purpose.

Due to this high redundancy, performance issues could arise. Possibly, STG could work more efficiently if duplicate transitions in the product of the specification and the test purpose were prevented.

3.4.2 Insufficient Closure

Test cases are intended to react promptly to inputs from the IUT. STG's way to ensure this, is to require input-completeness of the test case. In some cases, internal actions are hiding input actions. In order to generate test cases which are input-complete, all internal actions have to be removed.

Test Purpose	# Transitions incl. Duplicates	# Duplicate Transitions	Redundancy
1 answer_leave_tp	360	100	27.78 %
2 answer_pdu_tp	232	61	26.29 %
3 double_msgs_tp	314	82	26.11 %
4 join_tp	154	49	31.82 %
5 join_leave2_tp	378	102	26.98 %
6 join_leave_tp	433	127	29.33 %
7 join_leave_join_tp	510	162	31.76 %
8 receive_msg2_tp	879	271	30.83 %
9 send_msg2_tp	953	298	31.27 %
10 send_msg_tp	805	237	29.44 %
Sum	5018	1489	–
Average	502	149	29.67 %

Table 3.2: This table presents statistics about redundant transitions in the closed products from the Conference Protocol specification and ten different test purposes.

As already mentioned in Section 3.1.3, experiments with the SIP Registrar specification (see Section 6.2) and the Conference Protocol specification (see Section 6.3) showed that STG is not always able to eliminate all internal actions during the closure procedure. Hence, it is possible that the generated test cases do not always react promptly to inputs from the IUT.

3.4.3 Missing Determinization

The determinization operation is not yet implemented in the current version of STG. Hence, the generated test cases may be non-deterministic and verdicts may depend on the internal choices of the tester (see also Section 3.1.4).

3.4.4 Memory Errors

While experimenting with the STG tool and the Session Initiation Protocol (SIP, see Section 6.2), severe problems have been encountered. STG was not able to generate test cases for the specification of a SIP Registrar. Three main problems occurred during test case generation:

- Two test purposes caused a stack overflow (see Listing 3.4).
- The test case generation processes for three test purposes were aborted due to an error regarding buffer growth (see Listing 3.5).
- For one test purpose, STG exceeded the limit of 2.5 GB of RAM.

Due to the output shown in Listing 3.4 and Listing 3.5, it can be assumed that the main problems lie within the first reachability analysis phase performed by the tool NBac.

The SIP Registrar specification is large compared to previous examples used with STG (see Table 6.3). It consists of 24 locations, 52 transitions, 28 variables, and 2 system parameters although the SIP Registrar is just one part of the whole Session Initiation Protocol and a lot of abstractions have been performed. The bigger part of the transitions are guarded by long Boolean expressions. Listing 3.6 shows an example of a transition in the SIP Registrar specification with a guard of this complexity. The Conference Protocol specification is a rather large specification as well (see Table 6.5). It consists of 15 locations, 34 transitions, 15 variables, and no system parameters.

```
Reachability analysis phase:
  converting iosts ../../sip_closure to auto ...
  DONE => ../../sip.aut
Fatal error: exception Stack_overflow
```

Listing 3.4: Stack overflow error produced by STG during test case generation for the SIP Registrar.

```
Fatal error: exception Failure(''Buffer.add: cannot grow buffer'')
[SNIP]
Reachability analysis: nbac2auto.opt command failed
```

Listing 3.5: Buffer error produced by STG during test case generation for the SIP Registrar.

```
from InitialSecondRequest
  if(userId = request_userId and callId = request_callId and branch =
     request_branch + 1 and cSeq > request_cSeq and hasAuthentication =
     true and not(addr1 = missing and exp1 <> missing) and not(addr2 =
     missing and exp2 <> missing) and not(addr1 = missing and addr2 <>
     missing) and (not(user1_addr1 <> missing and user1_addr2 <> missing)
     or ((user1_addr1 = addr1 or user1_addr2 = addr1 or addr1 = missing
     or addr1 = asterisk) and (user1_addr1 = addr2 or user1_addr2 = addr2
     or addr2 = missing or addr2 = asterisk))) and (not((user1_addr1 =
     missing or user1_addr2 = missing) and not(user1_addr1 = missing and
     user1_addr2 = missing)) or addr2 = missing or addr1 = user1_addr1 or
     addr1 = user1_addr2 or addr2 = user1_addr1 or addr2 = user1_addr2))

  sync pin?(userId, callId, branch, cSeq, hasAuthentication, addr1, exp1,
      addr2, exp2, exp)

  do { request_userId := userId |
       request_callId := callId |
       request_branch := branch |
       request_cSeq := cSeq |
       request_hasAuthentication := hasAuthentication |
       request_addr1 := addr1 |
       request_exp1 := exp1 |
       request_addr2 := addr2 |
       request_exp2 := exp2 |
       request_exp := exp }
to InitialSendToCore;
```

Listing 3.6: A transition in the Session Initiation Protocol having a complex guard.

Unlike SIP, the Conference Protocol (see Section 6.3) can be handled by STG. Since the Conference Protocol does not contain guards which are as complex as the ones in the SIP Registrar specification, it could be guessed that the main problems lie within the complicated Boolean expressions of the guards. This sounds reasonable, since STG uses BDDs, which are limited in their applicability due to their memory requirements. In the worst case, BDDs grow exponentially with the size of the Boolean formula they are representing. Hence, the most common failure related to BDDs is "running out of memory" [17]. This conforms to our experiences.

4 New Approach using Symbolic Execution and SAT Solving

As already mentioned in Section 3.4, several problems were encountered while experimenting with STG:

- Potentially, the product operation produces redundant transitions.
- The closure operation is not always able to eliminate all internal actions.
- The determinization operation is not implemented.
- Presumably, large guards lead to severe problems during the reachability analysis.

Solving all of these problems in one master's thesis would go over the top. Hence, our approach is designed to sort out the last problem of the above list: Due to a severe problem during reachability analysis, STG was not able to generate test cases for one of our case studies: the SIP Registrar (see Section 6.2), which is of industrial relevance. The test case generation did not succeed for any of the used test purposes and terminated with a stack overflow error. As already mentioned in Section 3.4, it is supposed that this problem is caused by too large Boolean expressions in the guards of the transitions. The goal of the approach, which will be presented throughout this chapter, is to be able to generate test cases for the SIP Registrar specification.

4.1 Relation to STG

To refer to Figure 3.1 again, the user's manual work of modelling specifications and designing test purposes will stay the same. Inputs given to STG can serve as inputs for the new approach as well. The middle part depicted in light orange, i.e., the test case generation, will be changed. In general, it is possible to reuse STG's test execution process, which is depicted in dark orange at the bottom of the graphic. However, since our approach does not generate test cases in the form of IOSTS, this proceeding would require a transformation of our test cases into IOSTS. Nevertheless, the development of a new test execution framework is less elaborate than the transformation of our test cases into IOSTS. At the moment of writing this thesis, an appropriate execution framework is under development at Graz University of Technology.

Since STG's test case generation works fine before the first reachability analysis is performed, certain parts of STG will be reused by our approach. To recap, the test generation steps of STG are (cf. Section 3.1):

1. completion of the test purpose
2. product of the specification and the test purpose
3. closure of the product
4. determinization of the closed product (not implemented)
5. test case selection:
    (a) first reachability analysis
    (b) coreachability analysis
    (c) second reachability analysis
6. input completion and transformation to Java

The first three steps, i.e., completion, product, and closure, will be reused from STG. Hence, our approach is based on the same model (IOSTS, see Section 2.2). Furthermore, it uses the same conformance testing theory (see Section 2.3). The great benefit of reusing the concepts and interfaces of STG is that the same inputs can be used again. Thus, already defined system specifications and test purposes can be directly reused.

The problematic part of STG beginning at the test case selection via reachability and coreachability analysis will be replaced by symbolic execution and a test case selection algorithm based on the resulting symbolic execution tree.

Figure 4.1 shows the differences between STG's approach and our idea. Parts of STG which are reused are depicted in black and white. STG's test case selection algorithm with reachability and coreachability analyses is not part of our approach. Hence, it is semi-transparent and crossed out. It is replaced by a new test case selection procedure depicted in green and black.

Due to the different techniques used for test case selection, the obtained results are different and not directly comparable. Test cases generated by STG are IOSTS, whereas test cases generated via symbolic execution are trees (see Section 4.4.1). This implies two things:

1. Test cases generated by STG may contain loops. In contrast, test cases generated by our approach do not contain any loops, since they are trees.
2. The test case execution framework implemented by STG cannot be directly reused for test cases generated by our approach. In order to utilize STG's execution procedure, a transformation into IOSTS would be necessary. In Figure 4.1, the possible retransformation is indicated by a blue arrow. However, this transformation has not been implemented since it is more elaborate than the development of a simple test execution procedure that is directly applicable to our test cases.

4.1.1 Interfacing with STG

As can be seen in Figure 4.2, we do not directly work on STG's source, which is written in Objective Caml (OCaml)[1]. The new test case selection is implemented in Java (for further details see Chapter 5). Hence, an interface between STG's output and our implementation had to be found. There exist only two possibilities of saving IOSTS, which are internally used by STG (see STG Reference Manual[2]):

1. A `dot` file for an IOSTS can be generated via STG's *show* command.
2. A `java` test case can be generated from the IOSTS by using STG's *tojava* command.

The `dot` format contains a good representation of the structure of the IOSTS, which can be parsed automatically. However, it does not include a declaration of the data of the IOSTS. Hence, the data types of the variables, messages, and system parameters are missing. This lack of information can be compensated by the `java` test case implementation file, which contains a declaration of variables and parameters. The data types of the messages can be determined by parsing the action signatures. As can be seen in Figure 4.2, the interfacing of STG and our implementation requires both files, the `dot` and the `sjava` file of the IOSTS representing the closed product of the specification and the test purpose.

Since it is no great extra effort, we eliminate redundant transitions during parsing of the above mentioned files. Depending on the input specification, we can thus improve the performance of our approach, e.g., about 30 % of the Conference Protocol's transitions are redundant (see Section 3.4). The potential of saving calculation effort during symbolic execution (see Section 4.3.2) is especially high if the used specification contains loops in which many transitions are redundant and no state inclusion can be identified.

[1]`http://caml.inria.fr/ocaml/` (last visit 2009-09-27)
[2]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_4.html` (last visit 2009-09-27)

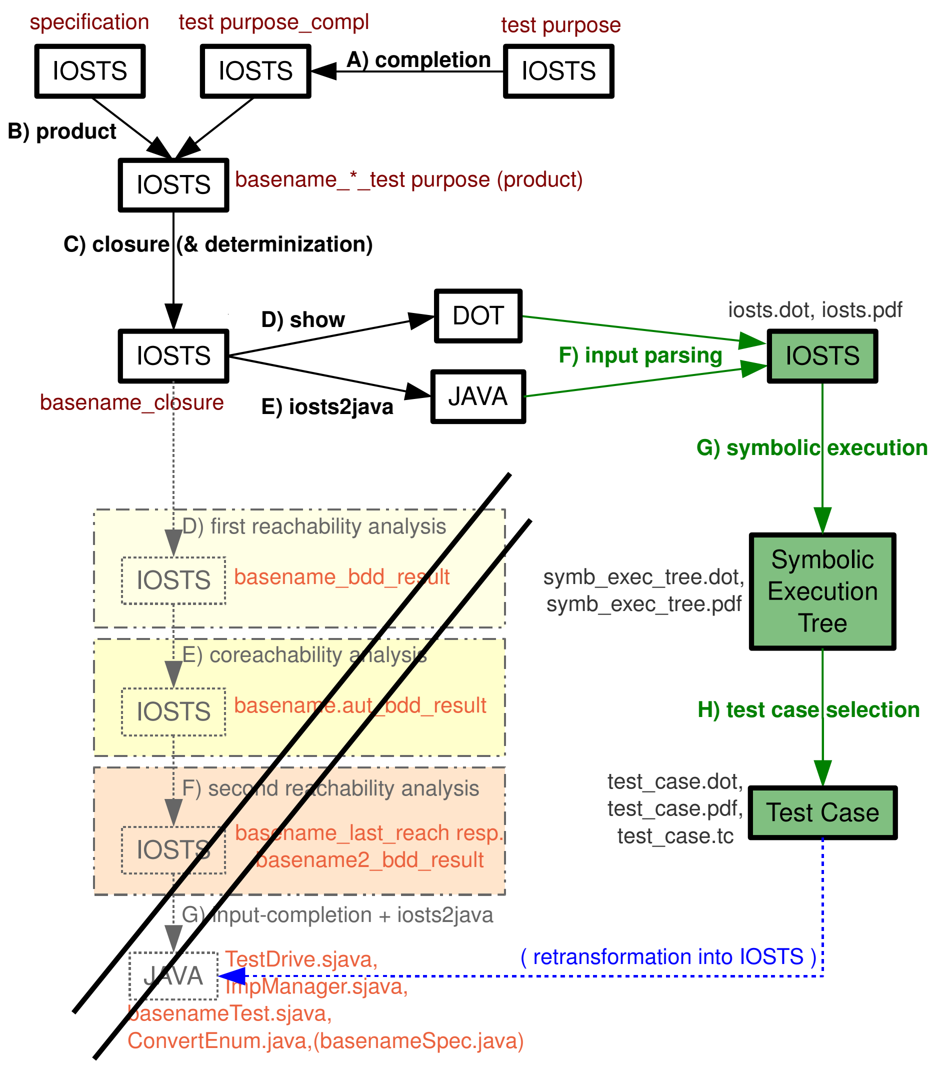


Figure 4.1: Comparison of our test case generation process to the one of STG: Although our test case generation process reuses certain parts of STG, it differs in the test case selection procedure. Reused parts of STG are depicted in black and white. STG's test case selection algorithm with reachability and coreachability analyses is semi-transparent and crossed out, since it is not part of our test case generation approach. It is substituted by a new test case selection procedure depicted in green and black.

4.2 Overview of the New Test Case Generation Approach

The main steps of our test case generation approach are:

1. completion of the test purpose
2. product of the specification and the test purpose
3. closure of the product
4. symbolic execution
5. test case selection

As depicted in Figure 4.2, the first three steps (completion, product calculation, and closure) are performed by STG (see Section 3.1 for further details). They are depicted in black and white, whereas newly implemented components are green and black. As already described in Section 4.1.1, we do

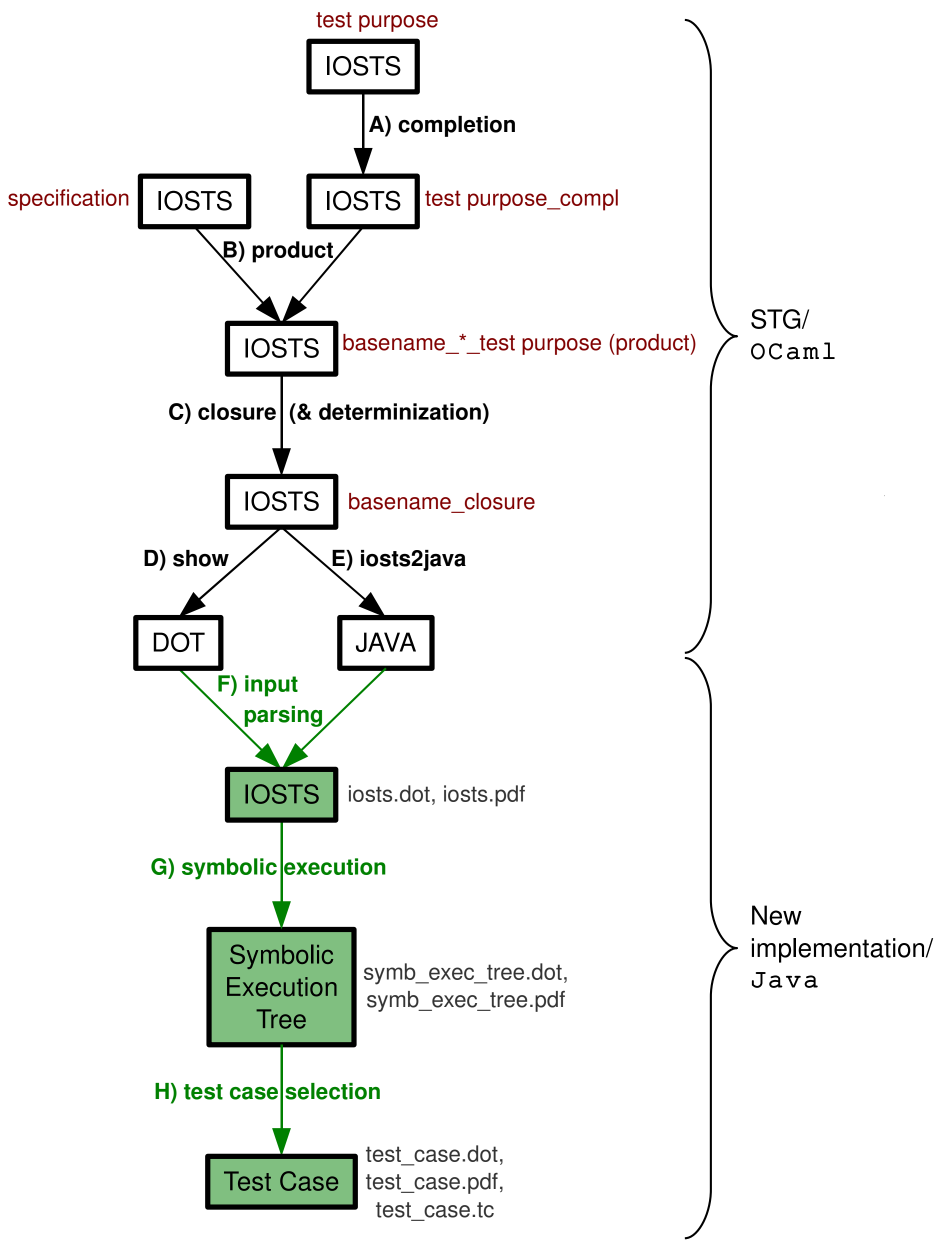


Figure 4.2: The new test case generation process is based on the foundations of STG. It reuses test purpose completion, product calculation, and closure. Parts implemented by STG are depicted in black and white. The test case selection is realized via symbolic execution and selection of certain paths of the resulting symbolic execution tree. It is depicted in green and black.

not work directly on STG's source. Hence, our implementation has to parse two output files generated by STG, which represent the closed product of the specification and the test purpose. This IOSTS is symbolically executed in order to generate a *symbolic execution tree* from which a test case is selected subsequently. The symbolic execution tree and the generated test case are written into a `dot` file and converted into `pdf` format via the Graphviz tool[3]. Additionally, the test case is exported into a specified text format in order to interface with a new test execution framework.

Symbolic execution (see Section 4.3) is a common methodology for test case generation. Although its main goal is to explore the possible execution paths of an application, it can be employed for test case generation, which can be seen as a side-effect of symbolic execution. By solving the constraints

[3]`http://www.graphviz.org` (last visit 2009-09-27)

of the path condition and providing the resulting values as input for the application, a program can be forced to follow a certain path. In this way, test cases can be constructed. More detailed information about symbolic execution of IOSTS and test case selection from symbolic execution trees will be given subsequently.

4.3 Symbolic Execution

Originally, symbolic execution was applied to programs (see Section 4.3.1). Recently, it has been adapted in order to be applicable on IOSTS (see Section 4.3.2).

4.3.1 Symbolic Execution of Programs

Symbolic execution, which is also referred to as symbolic evaluation or symbolic computation, is a technique for executing programs without concrete input values. Instead of using specific data, e.g., numbers, symbolic values are used as inputs. Hence, the calculation results of a symbolic execution differ from the ones obtained from normal program execution. In general, the outcome of a symbolic execution will be a term depending on the input symbols rather than a concrete value [53].

An approach similar to symbolic execution was already introduced in 1969 by Balzer [5]. Since then, a lot of research in the field of symbolic execution of programs has been conducted, e.g., by Boyer et al. [16], King [53; 54], Clarke [27], or Coward [29].

When talking about symbolic execution, it is necessary to clearly distinguish between *symbols* and *symbolic variable names*. Symbols represent the program inputs, which cover any data external to a program including parameters, global variables, read statements, etc. Symbols have to be looked at in the static mathematical sense and are used to represent some unknown but already fixed input value. The value of a program variable may change during program execution [53].

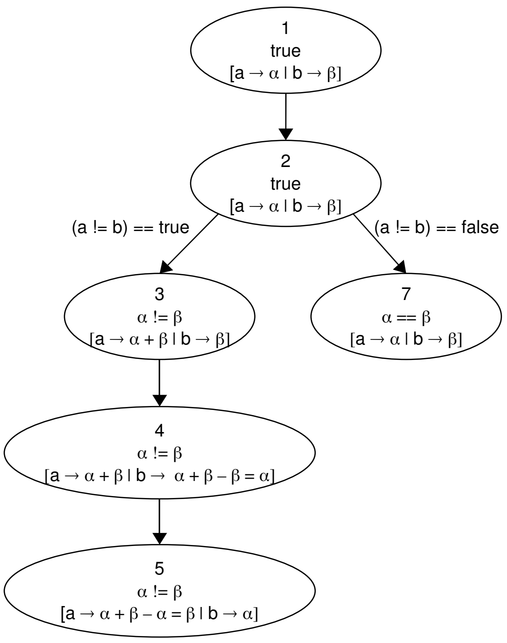
According to King [53], symbolic execution proceeds like normal program execution except when symbolic inputs are encountered. There are two basic scenarios, in which symbolic inputs have to be handled:

1. Computation of an expression involving symbols:
   Each programming language has different operators defined over data types, e.g., addition of integers, addition of floating point numbers, etc. These operators have to be extended in order to deal with symbolic values. This is easy for arithmetic operators by using the relationship between arithmetic and algebra. The value of a program variable X is denoted by $v(X)$. Consider a function with formal parameters A, B and symbolic input values $v(A) = \alpha$ and $v(B) = \beta$. Then, the symbolic execution of the statement $C = A + 2 * B$ would result in $\alpha + 2 * \beta$. At least in theory, it is possible to do similar generalizations for all computational operators defined for a programming language.

2. Conditional branching dependent on symbols:
   Consider the IF statement of the form IF B THEN S_1 ELSE S_2, where B is some Boolean expression and S_1 and S_2 are some statements. When executing a program normally, it is clear whether S_1 or S_2 has to be executed. This is not always the case in symbolic execution. $v(B)$ could be some expression over the input symbols. In this case, it is not always possible to decide whether $v(B)$ or $\neg v(B)$ evaluates to $true$. Since both scenarios are possible, the execution forks into two parallel executions. One assumes $v(B)$, the other one $\neg v(B)$. These assumptions are called path conditions. Each path of an execution has its own path condition. If one of the parallel executions reaches another conditional statement, another execution split may be required.
   At each conditional statement IF B THEN S_1 ELSE S_2, the path condition pc has to be updated by $pc = pc \wedge v(B)$ or $pc = pc \wedge \neg v(B)$ respectively. The path condition is an accumulator of conditions on symbolic inputs and determines a unique control path through the program.

```
1  int a, b;
2  if(a != b) {
3     a = a + b;
4     b = a - b;
5     a = a - b;
6  } else {
7     print("Nothing to do...");
8  }
```

(a) A program for swapping the values of two integers without using an intermediate variable.



(b) The symbolic execution tree for the program that swaps two integer values.

Figure 4.3: An example of a symbolic execution tree.

In some cases, the path condition pc before an IF statement can help to determine whether the condition B of this IF statement evaluates to $true$ or to $false$. Under the condition that $pc \neq false$, either $pc \rightarrow v(B)$ or $pc \rightarrow \neg v(B)$ is $true$. Hence, the symbolic execution does not need to fork again. If the implication $pc \rightarrow v(B)$ evaluates to $true$, then only the branch with statement S_1 is followed. If the implication $pc \rightarrow \neg v(B)$ evaluates to $true$, then only the branch with statement S_2 is followed.

The symbolic execution of a program can be represented by a tree, the so-called symbolic execution tree. The nodes of this tree are called execution states, which are triples consisting of the program statement counter, the path condition, and the variable values. The execution states are connected via directed arrows, which represent the transitions between the corresponding program statements. Each conditional statement, which forked the execution into several parallel executions, has more than one outgoing arrow labelled with the corresponding path choices. If the symbolically executed program contains a loop whose number of iterations depends on symbolic inputs, the symbolic execution tree is infinite [53].

Figure 4.3 gives an example to illustrate symbolic execution trees. Figure 4.3a shows a short program that swaps the values of two integers without using an intermediate variable. Figure 4.3b shows the corresponding symbolic execution tree. Each node has the following structure: The first line states the statement counter, the second line represents the path condition, and the third line shows the variable values. The variable values are enclosed by square brackets whereas the single values are divided by "|".

Symbolic execution is used for different purposes, e.g., verification, testing, debugging, program optimization, and program development. A survey about the various applications of symbolic execution was carried out by Clarke and Richardson [28].

4.3.2 Symbolic Execution of IOSTS

The technique of symbolic execution of programs has already been presented in Section 4.3.1. Gaston et al. [40] introduce symbolic execution for Input Output Symbolic Transition Systems (IOSTS, see Section 2.2). The main idea is the same as for symbolically executing programs. Concrete values of action messages as well as initialization values for IOSTS variables are replaced by symbolic values, which have unique names. Constraints on these variables are computed, which are named *path conditions*.

Similar to the symbolic execution of programs (see Section 4.3.1), a symbolic execution tree is built while symbolically executing an IOSTS. According to Gaston et al. [40], it consists of symbolic extended states (cf. execution states for programs), which are the vertexes of the tree, and symbolic transitions labelled by symbolic communication actions, which represent the tree's edges (cf. transitions between program statements):

Symbolic Extended State A symbolic extended state (SES) in a symbolic execution tree of an IOSTS $\mathcal{I} = \langle D, \Theta, Q, q_0, \Sigma, \mathcal{T} \rangle$ with data $D = V \cup P \cup M$ is a triple $\eta = \langle q, \pi, \sigma \rangle$. q identifies the corresponding location in the IOSTS $\mathcal{I}$ and conforms to the statement counter of an execution state for a program. Just like in the symbolic execution tree of a program, π is called path condition. σ is a mapping from $\mathcal{I}$'s variables and messages to their symbolic values (formulae). Hence, σ corresponds to the variable values, which are part of each execution state of a symbolic execution tree for a program. An SES is satisfiable if its path condition π, which is a Boolean expression, is satisfiable.

Symbolic Transition A symbolic transition is a triple $\langle \eta, sa, \eta' \rangle$. It is a connection between symbolic extended states. η is the source SES, η' the destination SES. A symbolic transition is labelled by a symbolic communication action sa.

Symbolic Communication Action A symbolic communication action is a tuple $sa = \langle a, \mu_{sa}, \sigma_{sa} \rangle$. a contains information about the IOSTS action from which the symbolic communication action is derived. This information includes the action's name and type. Each symbolic communication action has its own list of action messages (μ_{sa}), which consists of unique identifiers. Additionally, a mapping σ_{sa} from the original action's message names in μ to the symbolic communication action's unique message names in μ_{sa} is maintained.

By now, the components of a symbolic execution tree for an IOSTS have been introduced. The algorithm for the calculation of such a tree from IOSTS is based on the algorithm introduced by Gaston et al. [40]. However, they use a slightly different IOSTS model.

The discrepancy between the two IOSTS models concerns the use of messages as arguments for the actions. The IOSTS definition used in this work (see Section 2.2) and by STG does not support the use of variables, system parameters, or terms thereof as arguments of any actions. Hence, temporary variables only visible in a certain transition, so-called messages, have to be used. By contrast, Gaston et al. [40] do not support messages. Instead, the IOSTS variables and system parameters have to be used directly as arguments of the actions. Input actions carry variables and directly store the received input into these variables. Output actions pass certain values to the environment by using terms consisting of variables and system parameters.

To compensate this difference, the original algorithm by Gaston et al. [40] had to be adapted. In the following, the algorithm for executing an IOSTS with messages will be presented. Subsequently, modifications regarding the original algorithm will be discussed.

Modified Symbolic Execution Algorithm

The algorithm for calculating a symbolic execution tree for an IOSTS $\mathcal{I}$ with data $D = V \cup P \cup M$ is:

1. At first, the initial SES $init$ is calculated. It is a tuple $init = \langle q_0, true, \sigma_0 \rangle$ consisting of:

   - q_0, which is the initial location of the IOSTS $\mathcal{I}$.
   - $true$, which is the initial path condition.
   - σ_0, which is the initial σ and chosen arbitrarily. It maps each data item in the set $V \cup M$ to a new unique variable, which represents its symbolic value. In the following, the set of unique variables in the symbolic execution tree will be called F. Each newly created identifier id must neither be in the set D, nor in the current set of unique variables F. Hence, $id \notin D$ and $id \notin F$ is true at the point of creation. Afterwards, id will be added to F. The creation of unique identifiers for the system parameters P is unnecessary, because no new value can be assigned to a constant during execution of the IOSTS. System parameters can be seen as symbolic values, which do not change throughout execution. They have to be considered in test case execution by replacing them with their initially assigned values.

   The initial SES $init$ will be added to the set S, which is initially empty. S contains the symbolic extended states that have to be executed in the future.

2. Some SES $\eta = \langle q, \pi, \sigma \rangle$ is chosen from the set S and removed from S.

3. All IOSTS transitions originating from the location q of the chosen SES η are symbolically executed to calculate the corresponding symbolic transitions and their destination symbolic extended states. For each IOSTS transition $t = \langle q, a, \mu, G, A, q' \rangle$ originating from q, the following steps are executed:

   (a) The symbolic communication action which labels the symbolic transition st corresponding to the IOSTS transition t is denoted by sa. It is a tuple $sa = \langle a, \mu_{sa}, \sigma_{sa} \rangle$ where:

   - a is the IOSTS transition's action.
   - μ_{sa} denotes the list of symbolic messages. It is created by generating a new unique name $id \in F$ for each message of a. Hence, it contains only unique variables that are neither contained in D nor in F yet.
   - σ_{sa} maps the messages in μ to their corresponding symbolic messages in μ_{sa}.

   (b) The destination SES η' of the currently calculated symbolic transition st is $\eta' = \langle q', \pi', \sigma' \rangle$. It consists of:

   - q', which is the IOSTS location of η'. It is the destination location of the symbolically executed transition t.
   - π', which is the path condition of η'. It is the conjunction of the path condition of the source SES η and the guard of the executed IOSTS transition t in which each variable and message name has been substituted by its final unique identifier, i.e., $\pi' = \pi \wedge \sigma(\sigma_{sa}(G))$. Note that the application of mappings like σ and σ_{sa} to an expression like G substitutes all identifiers $x \in A$ (variables and messages of the IOSTS) that occur in the given expression by their corresponding symbolic values in the applied mapping (σ or σ_{sa} respectively). Hence, the application of a mapping to an expression returns an expression again.
   - σ', which contains the symbolic values for each $x \in V \cup M$ in η'. The symbolic values for each $x \in V \cup M$ can be expressed as $\sigma \circ \sigma_{sa} \circ A(x)$, whereas the assignments A can be interpreted as a function that returns for each $x \in V \cup M$ its assigned expression. $f \circ g$ denotes function composition of f and g. Hence, at first each identifier $x \in V \cup M$ is mapped to its value assigned by A, which may be an expression. Afterwards, each identifier $x \in V \cup M$ in the obtained expression will be substituted by its mappings in σ_{sa}. Finally, each identifier $x \in V \cup M$ in the last obtained formula will be substituted by its mapping in σ.

   (c) Thus, the calculated symbolic transition st is the triple $st = \langle \eta, sa, \eta' \rangle$.

(d) The destination SES η' is added to the set S and hence will be symbolically executed in the future if it satisfies the following requirements:

- Its IOSTS location q' is neither an *Accept* nor a *Reject* location. *Accept* and *Reject* locations always indicate the end of an execution path, since *Accept* means that the goal of the test purpose has been reached and *Reject* indicates that this behaviour shall not be tested by the used test purpose.
- It is satisfiable. If it is not, all symbolic extended states reachable from η' will not be satisfiable and hence not reachable.
- It is not included in another already calculated SES. State inclusion has to be considered, because symbolic executions of IOSTS with cycles would not terminate without an inclusion criterion. State inclusion will be explained below.
- Its IOSTS location q' has not been visited too often in the current path of the symbolic execution tree. An upper bound is necessary, since state inclusion is not always sufficient to guarantee termination. As already mentioned in Section 4.3.1, programs that contain loops whose number of iterations depends on symbolic inputs have an infinite symbolic execution tree.

If all IOSTS transitions originating from q have been symbolically executed, the symbolic execution of the SES η is finished. η is said to be symbolically executed.

4. The algorithm continues at Step 2 if there exists an already calculated SES that has not been executed yet, i.e., if the set S contains an SES that has not been executed yet.

Modifications

As already described above, the IOSTS model defined by Gaston et al. [40] does not support messages. Instead, IOSTS variables, system parameters, and terms thereof are used directly as arguments of the actions. Since STG is not compatible with this approach, the following modifications to the original algorithm by Gaston et al. [40] were necessary:

1. Gaston et al. create new identifiers only for the arguments of input actions. Output actions pass values to the environment via terms consisting of variables and system parameters, which do not need to be renamed, since no data is changed. In contrast, STG requires the use of messages as arguments of all actions. These messages are restricted to the desired output values by the guard. Since messages are local to the affected transition, message names can be reused in other transitions. In order not to mess up messages of different transitions in the path condition, output actions need to be handled like input actions: For each message of the action, a new identifier $id \in F$ needs to created.
2. The second modification affects the calculation of the path conditions and is caused by the first change. The generation of new message names for output actions as well as for input actions requires to take into account σ_{sa} in the calculation of π'. Gaston et al. calculate the path condition π' of the destination SES η' according to the formula $\pi' = \pi \wedge \sigma(G)$. When new message names are generated for all kinds of actions, the path condition has to be calculated by $\pi' = \pi \wedge \sigma(\sigma_{sa}(G))$ in order to use the current symbolic value of each message.

State Inclusion

Gaston et al. [40] motivate the use of a state inclusion criterion by the need of reactive systems to continuously communicate with their environments. In most cases, specifications of reactive systems contain internal loops. Without the state inclusion criterion, the symbolic execution of an IOSTS with internal loops would never terminate. Often, infinite behaviours are just sequences of “basic” behaviours.

If there exists an SES that comprises another one, there is no need to execute both. The symbolic execution of the included SES would be redundant.

Faivre and Gaston [35] base their state inclusion criterion on the so-called *symbolic state semantics* of an SES. Given an SES $\eta = \langle q, \pi, \sigma \rangle$, its symbolic state semantics is a tuple $Sem = \langle q, c^A \rangle$ consisting of

- q, which is the IOSTS location of the SES.
- c^A, which represents the constraints associated to the set A containing the variables and messages of the IOSTS. It is defined as the conjunction of the path condition π with all equations of the form $x = \sigma(x)$ for all $x \in A$. Formally:

$$c^A \;=\; \pi \;\wedge\; \bigwedge_{x \in A} (x = \sigma(x))$$

An SES $\eta_1 = \langle q_1, \pi_1, \sigma_1 \rangle$ with symbolic state semantics $Sem_1 = \langle q_1, c_1^A \rangle$ is included in another SES $\eta_2 = \langle q_2, \pi_2, \sigma_2 \rangle$ with symbolic state semantics $Sem_2 = \langle q_2, c_2^A \rangle$ if:

- $q_1 = q_2$ and
- c_1^A is stronger than c_2^A. More formally said, the implication $c_1^A \rightarrow c_2^A$ has to be a tautology.

Satisfiability and Tautology Checking

To find out whether an SES $\eta = \langle q, \pi, \sigma \rangle$ is satisfiable, it has to be determined whether its path condition π is satisfiable. However, the problem of deciding about the satisfiability of Boolean formulae is NP-complete [46]. Nevertheless, recent advances in SAT (Boolean satisfiability) solving have yielded powerful SAT solvers. In most cases, they can efficiently handle problems with up to millions of clauses and variables [33].

STG supports only two data types in its current version: Boolean and integer. Hence, the path condition π is a Boolean expression over identifiers and constants of type Boolean and/or integer, which may include arithmetic operators (see STG Reference Manual[4]). SMT (Satisfiability Modulo Theories) solvers are an extension to SAT solvers, which support arithmetics and other first-order theories like uninterpreted functions, arrays, or recursive datatypes [33]. Hence, for our application, SMT solvers are better suited than pure SAT solvers, since they allow the direct use of arithmetics in Boolean formulae.

For the calculation of state inclusions, it has to be found out whether the constraints associated to the variables and messages of the one state (c_1^A) are stronger than the constraints of the other state (c_2^A). Hence, it has to be determined, whether the implication $c_1^A \rightarrow c_2^A$ is a tautology or not. Again, this check can be accomplished by using an SMT solver. Obviously, a formula is a tautology if its negation is unsatisfiable. Hence, the SMT solver can be used to decide whether a formula is a tautology by checking whether the negated formula is unsatisfiable.

Termination and Loops

As already mentioned in Section 4.3.1, programs that contain loops whose number of iterations depends on symbolic inputs have an infinite symbolic execution tree. Generally, the same holds for IOSTS. However, state inclusion aims at preventing the symbolic execution from unnecessary loop unfolding. By checking if the semantics of a certain SES is already covered by the semantics of any other SES, the unfolding of a loop can be stopped. Nevertheless, this is not always possbile, since state inclusion checking requires SAT solving, which is known to be NP-complete. If the IOSTS contains an infinite loop that cannot be stopped via state inclusion, there is no chance to terminate at all.

[4]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_4.html` (last visit 2009-09-27)

In order to guarantee program termination independently from the IOSTS that is symbolically executed, an *upper bound* for loop unfolding is introduced. It is defined as the upper limit for the number of SES in any path of the symbolic execution tree that correspond to a certain IOSTS location. For example, an upper bound of two means that each path of the symbolic execution tree has at most two SES corresponding to the same IOSTS location.

Generally, the lower the value for the upper bound is chosen, the smaller is the resulting symbolic execution tree and thus, the faster the symbolic execution terminates. If it is too small, the symbolic execution may be terminated too soon and a test case selection, which requires at least one satisfiable *Accept* state, may not be possible (see Section 4.4.1).

4.3.3 Example: Triangle Type Checker

As already mentioned in Section 4.1, our approach for test case generation is based on the closed product generated by STG for a given specification and test purpose. For the Triangle Type Checker specification (Figure 2.2) and the "NotTriangle" test purpose (Figure 2.3), the closed product is depicted in Figure 3.8. Figure 4.4 shows the symbolic execution tree for the closed product. It was calculated according to the above described symbolic execution algorithm.

Each ellipse represents a symbolic extended state $\eta = \langle q, \pi, \sigma \rangle$. The first line in each ellipse consists of (1) an ID, which is a consecutive number, (2) the IOSTS location q, and (3) the number of visits of q, which states how many SES in the path leading to η (including η) have q as their IOSTS location. The following lines consist of η's path condition π and its σ, which maps each variable and message to its symbolic value. σ is enclosed by square brackets.

Consider the third SES in Figure 4.4. Its first line is "3 CheckPositive_S1 (1)", which means that its ID is "3", it refers to the IOSTS location "CheckPositive_S1", and "(1)" states that this IOSTS location is referred to for the first time in the current path of the symbolic execution tree. The second line contains "true", which is the path condition. The last two lines contain the symbolic values for the variables and messages that apply in this SES. All newly created identifiers that represent symbolic values are a concatenation of the original variable's name, the character "#", and a consecutive number. For example, "$b \rightarrow q$#1" says that the symbolic value of the variable b is q#1, whereas the symbolic value q#1 represents some value of the message q.

Arrows represent the symbolic transitions of the form $st = \langle \eta, sa, \eta' \rangle$, whereas each symbolic communication action $sa = \langle a, \mu_{sa}, \sigma_{sa} \rangle$ is depicted by a rectangle, in which the first part states the name a of the symbolic communication action, its type (input ?, output !), and its symbolic messages μ_{sa}. The second part represents the mapping σ_{sa} from the original message names in the IOSTS to the newly created symbolic messages. σ_{sa} is enclosed by square brackets.

Consider the transition between the second and the third SES of Figure 4.4. Its first line "Read!(p#1, q#1, r#1)" means that the symbolic communication action is named "Read" and that this action is an input action, which is indicated via an exclamation mark. The symbolic communication action has three symbolic messages, i.e., $\mu_{sa} = [p\#1, q\#1, r\#1]$. The second line representing the mapping σ_{sa} is "$[r \rightarrow r\#1 \mid q \rightarrow q\#1 \mid p \rightarrow p\#1]$", which means that the IOSTS message r is mapped to the symbolic value r#1, q is mapped to $q\#1$, and p is mapped to p#1. Again, identifiers containing a "#" denote symbolic values.

Note that locations in the closed product which are unreachable due to the structure of the IOSTS are not part of the symbolic execution. For example, the location *CheckTriangle_S1* in Figure 3.8 is not taken into account, since it has no predecessors and it is not the initial location of the IOSTS. Hence, the symbolic execution algorithm will never calculate an SES related to the location *CheckTriangle_S1*.

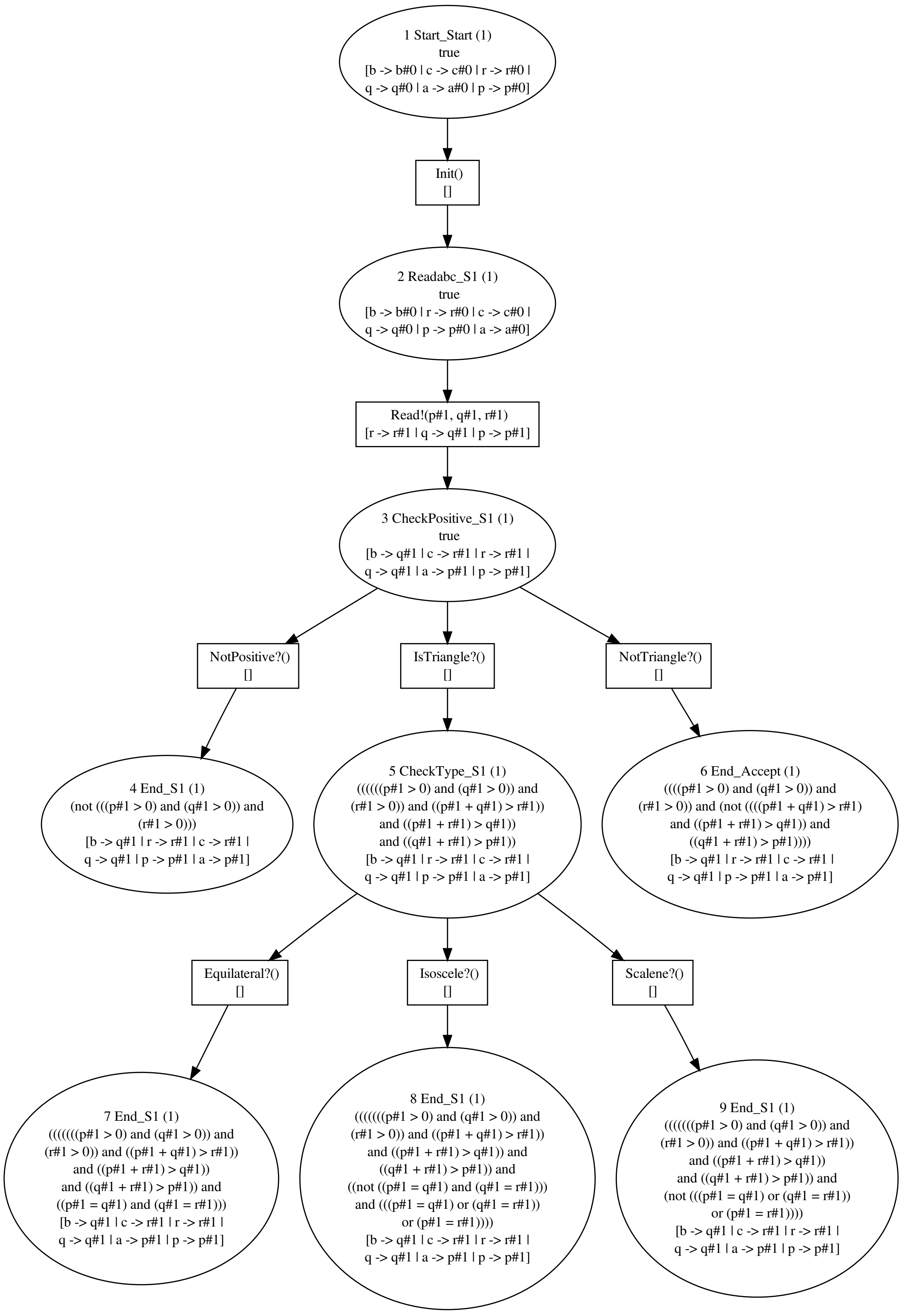

1 Start_Start (1)
true
[b -> b#0 | c -> c#0 | r -> r#0 | q -> q#0 | a -> a#0 | p -> p#0]
Init()
[]
2 Readabc_S1 (1)
true
[b -> b#0 | r -> r#0 | c -> c#0 | q -> q#0 | p -> p#0 | a -> a#0]
Read!(p#1, q#1, r#1)
[r -> r#1 | q -> q#1 | p -> p#1]
3 CheckPositive_S1 (1)
true
[b -> q#1 | c -> r#1 | r -> r#1 | q -> q#1 | a -> p#1 | p -> p#1]
NotPositive?()
[]
IsTriangle?()
[]
NotTriangle?()
[]
4 End_S1 (1)
(not (((p#1 > 0) and (q#1 > 0)) and (r#1 > 0)))
[b -> q#1 | r -> r#1 | c -> r#1 | q -> q#1 | p -> p#1 | a -> p#1]
5 CheckType_S1 (1)
((((((p#1 > 0) and (q#1 > 0)) and (r#1 > 0)) and ((p#1 + q#1) > r#1)) and ((p#1 + r#1) > q#1)) and ((q#1 + r#1) > p#1))
[b -> q#1 | r -> r#1 | c -> r#1 | q -> q#1 | p -> p#1 | a -> p#1]
6 End_Accept (1)
((((p#1 > 0) and (q#1 > 0)) and (r#1 > 0)) and (not ((((p#1 + q#1) > r#1) and ((p#1 + r#1) > q#1)) and ((q#1 + r#1) > p#1))))
[b -> q#1 | r -> r#1 | c -> r#1 | q -> q#1 | p -> p#1 | a -> p#1]
Equilateral?()
[]
Isoscele?()
[]
Scalene?()
[]
7 End_S1 (1)
(((((((p#1 > 0) and (q#1 > 0)) and (r#1 > 0)) and ((p#1 + q#1) > r#1)) and ((p#1 + r#1) > q#1)) and ((q#1 + r#1) > p#1)) and ((p#1 = q#1) and (q#1 = r#1)))
[b -> q#1 | c -> r#1 | r -> r#1 | q -> q#1 | a -> p#1 | p -> p#1]
8 End_S1 (1)
(((((((p#1 > 0) and (q#1 > 0)) and (r#1 > 0)) and ((p#1 + q#1) > r#1)) and ((p#1 + r#1) > q#1)) and ((q#1 + r#1) > p#1)) and ((not ((p#1 = q#1) and (q#1 = r#1))) and (((p#1 = q#1) or (q#1 = r#1)) or (p#1 = r#1))))
[b -> q#1 | c -> r#1 | r -> r#1 | q -> q#1 | a -> p#1 | p -> p#1]
9 End_S1 (1)
(((((((p#1 > 0) and (q#1 > 0)) and (r#1 > 0)) and ((p#1 + q#1) > r#1)) and ((p#1 + r#1) > q#1)) and ((q#1 + r#1) > p#1)) and (not (((p#1 = q#1) or (q#1 = r#1)) or (p#1 = r#1))))
[b -> q#1 | c -> r#1 | r -> r#1 | q -> q#1 | a -> p#1 | p -> p#1]

Figure 4.4: The symbolic execution tree for the IOSTS depicted in Figure 3.8.

4.4 Test Case Selection

STG uses reachability and coreachability analyses to select a test case from the closed product of a given specification and test purpose (see Section 3.1.5). By contrast, the approach of this work symbolically executes the closed product and subsequently generates a test case by selecting certain paths from the resulting symbolic execution tree. By the use of these different techniques, the generated test cases are different and not directly comparable. Test cases generated by STG are IOSTS that possibly contain loops. In contrast, test cases generated via symbolic execution are trees. Hence, our test cases do not contain any loops. Anyway, it is possible to transform test cases generated by our approach into IOSTS. Subsequently, the algorithm for selecting test cases from symbolic execution trees will be presented and illustrated with our running example.

4.4.1 Test Case Selection Algorithm

The algorithm for the generation of a test case from a symbolic execution tree is based on the idea of selecting accepted behaviour similarly to TGV [48]. Our algorithm selects one test case per test purpose and consists of the following steps:

1. An arbitrary, but satisfiable *Accept* state $\eta_A = \langle q_A, \pi_A, \sigma_A \rangle$ is chosen. The location q_A is an accepting state as defined by the used test purpose. If there is no satisfiable SES that corresponds to an *Accept* location, no test case can be generated.
2. The path leading from the root SES $init$ to the chosen *Accept* state η_A is added to the test case by backtracking from η_A, which is a leaf of the symbolic execution tree, to the root node $init$.
3. By now, the selected test case is one path of the symbolic execution tree. In this step, it is possibly extended to a tree by adding additional symbolic transitions and symbolic extended states. The algorithm for deciding which symbolic transitions and SES have to be added works top-down. Starting at the root SES $init$, it investigates the outgoing symbolic transitions of the symbolic extended states and decides whether they have to be added to the test case. Therefor, a list S is maintained, which contains all SES that have to be investigated in the future. Initially, S contains all SES that already belong to the test case due to Step 2 of the test case selection algorithm except for the leaf SES. Hence, one outgoing symbolic transition and one successor state of each SES in S is already part of the test case initially.

   In order to add a symbolic transition $st = \langle \eta, sa, \eta' \rangle$ and its destination SES η' to the test case, η' has to be satisfiable. If η' is part of the test case and it is no *Accept* or *Reject* state, then it is added to S. Hence, its outgoing symbolic transitions will be investigated. If all outgoing transitions of an SES η are checked, then η is removed from S. Step 3 loops until no more symbolic transitions have to be investigated, i.e., until S is empty. The exact rules for deciding which symbolic transitions and SES have to be added to the test case will be presented below.

   In the context of test case selection, *input* and *output* refer to the input/output of the tester (not of the system specification or the SUT respectively), since STG has already switched input actions to output actions and vice versa. The rules that define which outgoing symbolic transitions of an SES η have to be added to the test case are:

   I. If one of η's outgoing transitions labelled with an output action is already part of the test case, then no further outgoing transitions of η have to be added to the case.
   II. If one of η's outgoing transitions labelled with an input action is already part of the test case, then all outgoing transitions of η that are labelled by input, initial, or internal actions have to be added to the test case.
   III. If one of η's outgoing transitions labelled with an initial or internal action is already part of the test case, then all transitions of η that are labelled by input, initial, or internal actions have to be added to the test case.

IV. If none of η's outgoing transitions is part of the test case yet, then the following rules apply:
   (a) If η has outgoing transitions labelled by input actions, then all transitions of η that are labelled by input, initial, or internal actions have to be added to the test case.
   (b) If η does not have any outgoing transitions labelled by input actions, but it has outgoing transitions labelled by output actions, then one of η's transitions labelled with an output action has to be added to the test case.
   (c) If η does neither have transitions labelled by an input action nor transitions labelled by an output action, then all of η's transitions labelled with an initial or internal action have to be added to the test case.

To summarize the above rules, one SES in a test case may either have only inputs or only outputs as outgoing symbolic transitions. If possible, i.e., if no output is already part of the test case, inputs have priority. Symbolic transitions labelled with symbolic communication actions of type *init* or *internal* are only added if there are no outputs added to the test case at the current SES. In this way, controllability of the test case is ensured, i.e., choices between outputs or between inputs and outputs are prevented [48]. The reasons why initial and internal actions have to be taken into account are the following:

- STG requires each IOSTS to be initialized via an *init* action, which is a special action. It does not belong to any alphabet of the IOSTS (Σ^i, Σ^o, Σ^{int}). Hence, it has to be treated in a special manner.
- As already mentioned in Section 3.1.3, STG is not always able to eliminate all internal actions during closure. Hence, the symbolic execution tree may contain internal actions that have to be considered during test case selection.

Main Differences to TGV's Approach

Besides the fact that our algorithm is based on symbolic execution trees while TGV [48] works with IOLTS, the following differences can be identified:

- TGV offers two methods for test case generation: The first one creates a *complete test graph* (CTG) from which all test cases corresponding to the used test purpose are selected. The second way of test case generation works on the fly and produces just one test case. So far, our approach supports the generation of one test case per test purpose, but it is not able to generate all potential test cases. Hence, no CTG is generated.
- Test cases created by TGV include explicit verdicts. In contrast, tests generated by our approach contain implicit verdicts. There are no *Inconclusive* or *Fail* states in our test cases. Paths that do not lead to an *Accept* state are not pruned and substituted by an *Inconclusive* state, which indicates that the goal of the test purpose cannot be achieved any more. Verdicts for our test cases have to be made explicit during test case execution: If the test execution ends in an *Accept* state, the verdict is *Pass*. If it reaches an end state that is not an *Accept* state, i.e., the IUT behaved correctly although the goal of the test purpose could not be attained, then the verdict is *Inconclusive*. If the IUT sends some input to the test case that is not recognized in the current state, then the verdict is *Fail*.

4.4.2 Example: Triangle Type Checker

It is possible to generate a test case from the symbolic execution tree of the Triangle Type Checker shown in Figure 4.4, since it contains a satisfiable *Accept* state (*End_Accept*). After the selection of the path

$$\mathit{Start\_Start} \xrightarrow{Init} \mathit{Readabc\_S1} \xrightarrow{Read!} \mathit{CheckPositive\_S1} \xrightarrow{NotTriangle?} \mathit{End\_Accept},$$

the investigation whether other transitions have to be added to the test case is performed. Since there are no symbolic extended states where choices have to be made between several outputs or between inputs and outputs, the test case selection algorithm results in a test case that covers the whole tree. Hence, the symbolic execution tree and the generated test case are equivalent. They are depicted in Figure 4.4.

4.5 Benefits and Limitations

As already stated at the beginning of this chapter, the main goal of this work was to resolve the problems faced by STG during test case generation for the SIP Registrar (see Section 3.4). We were successful in overcoming this difficulty and have managed to generate test cases for the SIP Registrar (see Section 6.2). Nevertheless, when applied to the Conference Protocol (see Section 6.3), our approach suffers from performance problems and is not able to generate test cases for all specified test purposes.

The approach as it has been presented in this work has still room for improvement. At present, the algorithm for test case selection (see Section 4.4.1) is not very sophisticated. It generates just one test case by arbitrarily selecting one *Accept* state of the symbolic execution tree.

Another weakness of our approach is the missing guard strengthening. STG strengthens the guards during its coreachability phase in order to avoid that the test case follows a path that leads to an *Inconclusive* state (see Section 3.1.5). Although an inconclusive verdict cannot be prevented entirely, STG has better chances of reaching the goal of the test purpose than our approach has. Hence, the approach of this work is more likely to generate test cases that lead to an inconclusive verdict than the STG tool.

Since the determinization of IOSTS is not implemented in STG and this was not the main focus of this work, we cannot guarantee to generate test cases that do not depend on internal choices of the tester.

On the other hand, our approach does also provide several advantages. Besides the capability to generate test cases for industrial-sized specifications like the SIP Registrar (see Section 6.2), it generates test cases that do not contain any loops. Without loops, the test case execution framework does never have to decide when to stop looping. Under the condition that the IUT does not deadlock, we can guarantee that the test case execution results in a verdict for the executed test.

Our approach is designed to work with the same input as used for test case generation with STG. Hence, specifications and test purposes can be reused without modification. Since the modelling of large systems and the design of significant test purposes is a challenging task that cannot be entirely automated, the direct reuse of input files is a great advantage.

Another benefit of our work is the elimination of duplicate transitions in the product of the specification and the test purpose. STG processes IOSTS that possibly contain redundant transitions caused by the product operation. The resulting test cases are possibly larger than necessary. Our approach eliminates duplicate transitions and can thus increase its chances to successfully handle large specifications.

5 Design and Implementation

This chapter will present the prototype implementation of the new test case generation approach that was introduced in Chapter 4. It is intended to give an overview of the main aspects of the implementation and will not elaborate on each single class. For a more detailed description of the Java classes, we refer to the source code documentation in `Javadoc` format.

5.1 Overview

Figure 5.1 gives a rough overview of the prototype implementation of the new test case generation approach (see Chapter 4). It illustrates that the implementation of our approach splits in two main parts, which are integrated into one application by the use of a shell script. The first part is depicted in yellow and consists of STG's reused components that are implemented in OCaml. The second part is represented in green and comprises a new Java implementation.

At first, our approach uses STG to calculate the *closed product* of the two input IOSTS, which are a specification of the SUT and a test purpose in the form of an IOSTS like they are used for test case generation with STG. The closed product is calculated by (1) completion of the test purpose, (2) calculation of the product between the specification and the completed test purpose, and (3) closure of the product. The closed product is exported into `dot` and `java` format in order to interface with the Java part of this implementation. Details about STG's operations have already been given in Section 3.1. The interface between STG and the new Java implementation has already been discussed in Section 4.1.1.

The second part concerns our new Java implementation, which deals with three tasks. At first, the two files containing the closed product generated by STG are parsed. Subsequently, the closed product is symbolically executed like described in Section 4.3.2. Finally, a test case is selected from the resulting symbolic execution tree according to the algorithm introduced in Section 4.4.1. The final test case is exported into `dot`, `pdf`, and a specified text format (`tc`) in order to interface with the test execution framework, which is under development at the moment of writing this thesis.

Since STG has already been described in detail in Chapter 3, the rest of this chapter will be focused on the Java part implementing symbolic execution and test case selection from symbolic execution trees. In addition, a shell script for the integration of STG and the Java implementation into one application will be given.

5.2 Java Implementation

This section discusses the main aspects of the Java application, which was developed for Java 6. It represents the second part of the implementation of the new test case generation approach (cf. Figure 5.1) and implements two main functionalities, which are the symbolic execution of IOSTS and the test case selection from symbolic execution trees. Note that this program was written in order to process input that was automatically generated by STG. Since STG's output is supposed to be type-correct, our program does not perform any type checking.

5.2.1 IOSTS Input Parsing

IOSTS Parsers

As already described in Section 4.1.1, the only possibility to gather all needed information about an IOSTS, which is internally used by STG, is to export it into both `dot` and `sjava` format. The `dot` file contains information about the structure of the IOSTS, but it does not provide sufficient information

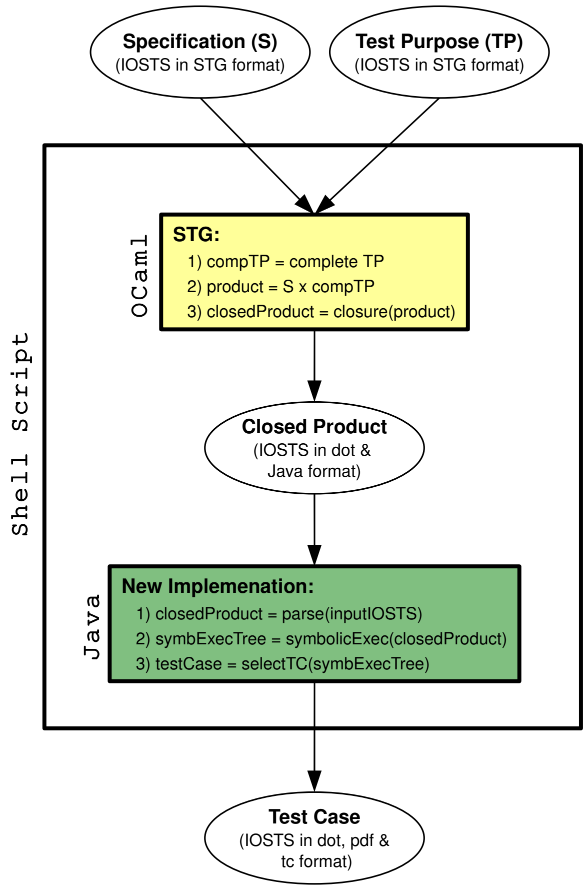

Specification (S)
(IOSTS in STG format)
Test Purpose (TP)
(IOSTS in STG format)
Shell Script
OCaml
STG:
1) compTP = complete TP
2) product = S x compTP
3) closedProduct = closure(product)
Closed Product
(IOSTS in dot & Java format)
Java
New Implemenation:
1) closedProduct = parse(inputIOSTS)
2) symbExecTree = symbolicExec(closedProduct)
3) testCase = selectTC(symbExecTree)
Test Case
(IOSTS in dot, pdf & tc format)

Figure 5.1: This figure gives a rough overview of the prototype implementation of the new test case generation approach. It illustrates that the implementation of our approach splits in two main parts, which are integrated into one application by the use of a shell script. The first part is depicted in yellow and consists of STG's reused components. The second part is represented in green and comprises a new Java implementation.

about its data. In order to determine the data types of the system parameters, variables, and messages of the IOSTS, the `sjava` file has to be parsed, too. Figure 5.2 shows the classes responsible for IOSTS parsing:

The *IostsParser* is an abstract class. It implements the `parseIosts` method, which reads the content of a text file into a String and calls the abstract `parse` method. Each derived class must implement this method with its own parser logic. Two classes are derived from the *IostsParser*:

The *IostsStructureParser* is responsible for parsing the `dot` file. It provides information about the locations and transitions of the IOSTS. The information about the transitions is split into two parts: The method `getTransitions` returns the source and destination location of each transition. The method `getTransitionInfos` returns action name and type as well as the guard and the assignments associated to the transitions. The *IostsDataParser* is responsible for parsing the `sjava` file. It provides information about the system parameters, variables, and action signatures.

The *IostsBuilder* invokes the `parseIosts` method on an instance of the *IostsStructureParser* and an instance of the *IostsDataParser*. It uses the information provided by the two parsers in order to build an IOSTS structure consisting of the classes shown in Figure 5.3.

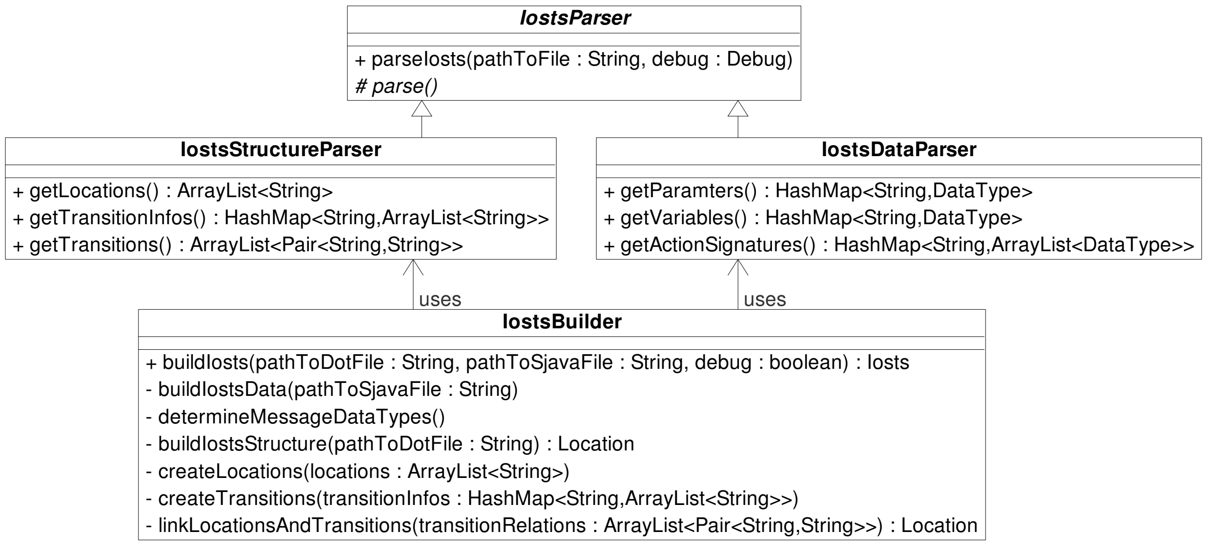

IostsParser
+ parseIosts(pathToFile : String, debug : Debug)
# parse()
IostsStructureParser
+ getLocations() : ArrayList<String>
+ getTransitionInfos() : HashMap<String,ArrayList<String>>
+ getTransitions() : ArrayList<Pair<String,String>>
IostsDataParser
+ getParamters() : HashMap<String,DataType>
+ getVariables() : HashMap<String,DataType>
+ getActionSignatures() : HashMap<String,ArrayList<DataType>>
uses
uses
IostsBuilder
+ buildIosts(pathToDotFile : String, pathToSjavaFile : String, debug : boolean) : Iosts
- buildIostsData(pathToSjavaFile : String)
- determineMessageDataTypes()
- buildIostsStructure(pathToDotFile : String) : Location
- createLocations(locations : ArrayList<String>)
- createTransitions(transitionInfos : HashMap<String,ArrayList<String>>)
- linkLocationsAndTransitions(transitionRelations : ArrayList<Pair<String,String>>) : Location

Figure 5.2: This class diagram shows the classes responsible for IOSTS parsing. The abstract class *IostsParser* has two subclasses: *IostsStructureParser* and *IostsDataParser*. The information provided by the two parsers is used by the *IostsBuilder* to construct an IOSTS data structure like depicted in Figure 5.3.

Data Structure for IOSTS

As shown in Figure 5.3, the IOSTS data structure is implemented straight forward. An *Iosts* consists of *Location*s which are connected via *Transition*s. Each transition has a start and destination location as well as a *Guard*, an *Action* that carries messages, and *Assignments*. An *Iosts* object and all of its components are built and assembled by the *IostsBuilder* that gets information about the IOSTS from the *IostsStructureParser* and the *IostsDataParser* (see Figure 5.2).

Expression Parser and Expression Trees

While the *IostsStructureParser* and the *IostsDataParser* are implemented by hand, the classes for parsing expressions like they are used for the guard of a transition or for the single assignments, were generated by using the *ANTLR Parser Generator* (version 3.0). In our application, each expression is represented by an *expression tree* [62]. In Figure 5.4, the classes generated by ANTLR from a simple grammar file are depicted in grey. White classes were implemented manually in order to provide additional functionality for expression trees. For example, our class *ExprTree* provides a method to construct conjunctions, implications, or equalities of two given expression trees. Additionally, it supports the conversion of the represented expression into various text formats, e.g., infix notation.

5.2.2 Symbolic Execution

Symbolic Execution Algorithm

Figure 5.5 shows the classes that are involved in the symbolic execution of an *Iosts* object. The *SymbolicExecutor* is the central class of this package and implements the symbolic execution algorithm as defined in Section 4.3.2 as a breadth-first search. It maintains a list named `statesToBeExecuted_` corresponding to the set S introduced in the algorithm. It contains all symbolic extended states (SES) that have to be executed in the future.

The initial *SymbolicExtendedState* $init$ is created according to Step 1 of the algorithm when a *SymbolicExecutor* is instantiated with an *Iosts* object. The actual symbolic execution (Step 2 to 4 of the

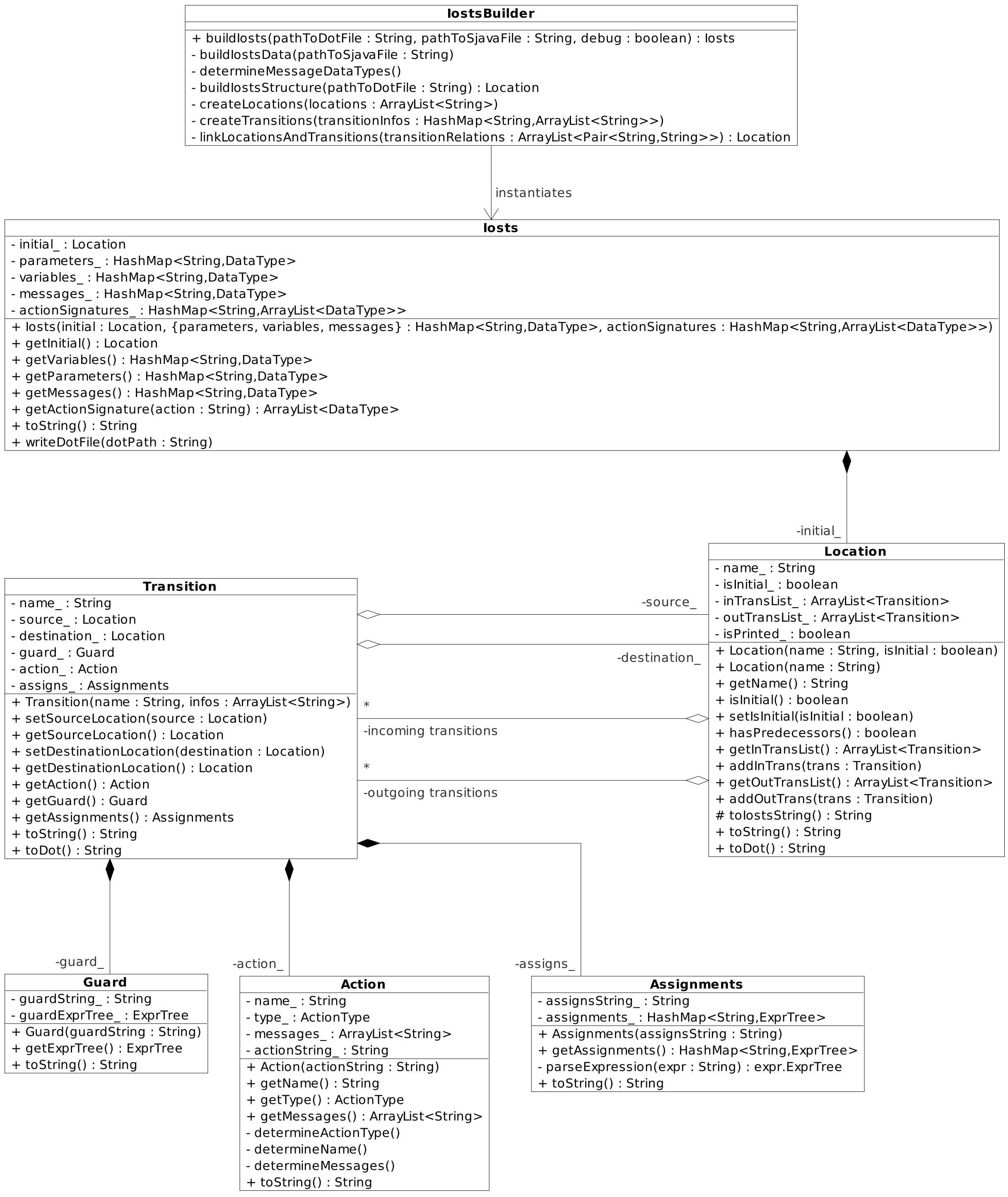

IostsBuilder
+ buildIosts(pathToDotFile : String, pathToSjavaFile : String, debug : boolean) : Iosts
- buildIostsData(pathToSjavaFile : String)
- determineMessageDataTypes()
- buildIostsStructure(pathToDotFile : String) : Location
- createLocations(locations : ArrayList<String>)
- createTransitions(transitionInfos : HashMap<String,ArrayList<String>>)
- linkLocationsAndTransitions(transitionRelations : ArrayList<Pair<String,String>>) : Location
instantiates
Iosts
- initial_ : Location
- parameters_ : HashMap<String,DataType>
- variables_ : HashMap<String,DataType>
- messages_ : HashMap<String,DataType>
- actionSignatures_ : HashMap<String,ArrayList<DataType>>
+ Iosts(initial : Location, {parameters, variables, messages} : HashMap<String,DataType>, actionSignatures : HashMap<String,ArrayList<DataType>>)
+ getInitial() : Location
+ getVariables() : HashMap<String,DataType>
+ getParameters() : HashMap<String,DataType>
+ getMessages() : HashMap<String,DataType>
+ getActionSignature(action : String) : ArrayList<DataType>
+ toString() : String
+ writeDotFile(dotPath : String)
-initial_
Location
- name_ : String
- isInitial_ : boolean
- inTransList_ : ArrayList<Transition>
- outTransList_ : ArrayList<Transition>
- isPrinted_ : boolean
+ Location(name : String, isInitial : boolean)
+ Location(name : String)
+ getName() : String
+ isInitial() : boolean
+ setIsInitial(isInitial : boolean)
+ hasPredecessors() : boolean
+ getInTransList() : ArrayList<Transition>
+ addInTrans(trans : Transition)
+ getOutTransList() : ArrayList<Transition>
+ addOutTrans(trans : Transition)
# toIostsString() : String
+ toString() : String
+ toDot() : String
-source_
-destination_
*
-incoming transitions
*
-outgoing transitions
Transition
- name_ : String
- source_ : Location
- destination_ : Location
- guard_ : Guard
- action_ : Action
- assigns_ : Assignments
+ Transition(name : String, infos : ArrayList<String>)
+ setSourceLocation(source : Location)
+ getSourceLocation() : Location
+ setDestinationLocation(destination : Location)
+ getDestinationLocation() : Location
+ getAction() : Action
+ getGuard() : Guard
+ getAssignments() : Assignments
+ toString() : String
+ toDot() : String
-guard_
-action_
-assigns_
Guard
- guardString_ : String
- guardExprTree_ : ExprTree
+ Guard(guardString : String)
+ getExprTree() : ExprTree
+ toString() : String
Action
- name_ : String
- type_ : ActionType
- messages_ : ArrayList<String>
- actionString_ : String
+ Action(actionString : String)
+ getName() : String
+ getType() : ActionType
+ getMessages() : ArrayList<String>
- determineActionType()
- determineName()
- determineMessages()
+ toString() : String
Assignments
- assignsString_ : String
- assignments_ : HashMap<String,ExprTree>
+ Assignments(assignsString : String)
+ getAssignments() : HashMap<String,ExprTree>
- parseExpression(expr : String) : expr.ExprTree
+ toString() : String

Figure 5.3: The IOSTS data structure is implemented straight forward. An *Iosts* consists of *Locations* which are connected via *Transitions*. Each transition has a start and destination location as well as a *Guard*, an *Action* that carries messages, and *Assignments*. The *Iosts* with all of its subcomponents is constructed by the *IostsBuilder* class.

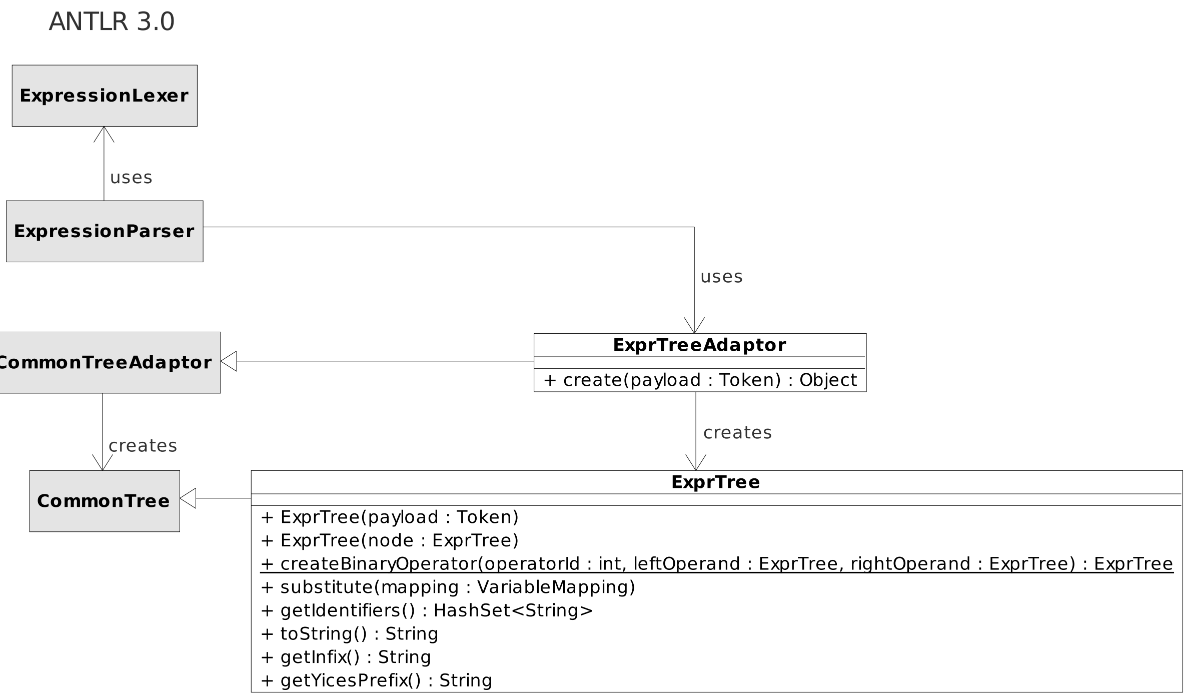

ANTLR 3.0
ExpressionLexer
uses
ExpressionParser
uses
CommonTreeAdaptor
ExprTreeAdaptor
+ create(payload : Token) : Object
creates
creates
CommonTree
ExprTree
+ ExprTree(payload : Token)
+ ExprTree(node : ExprTree)
+ createBinaryOperator(operatorId : int, leftOperand : ExprTree, rightOperand : ExprTree) : ExprTree
+ substitute(mapping : VariableMapping)
+ getIdentifiers() : HashSet<String>
+ toString() : String
+ getInfix() : String
+ getYicesPrefix() : String

Figure 5.4: The classes for parsing expressions and constructing expression trees were generated by the parser generator ANTLR 3.0.

algorithm) starts when the `execute` method is invoked. It symbolically executes each SES in the `statesToBeExecuted_` list with the aid of the `executeTransitions` method, which symbolically executes all transitions of the current SES. The calculation of the destination SES of the currently executed transition is encapsulated in the method `createDestinationSES`. The path condition of the destination SES is calculated with the method `createNewPC`. The mapping $\sigma : A \rightarrow F$, which maps all variables and messages to their symbolic values in the destination SES is created with the method `createNewSigma`. The *VariableGenerator* class is used to create unique identifiers $id \in F$, which serve as symbolic values of the variables and messages of the IOSTS. In order to guarantee that all created identifiers are unique, each *SymbolicExecutor* must not use more than one instance of the *Variable Generator* class (singleton). The `execute` method of the *SymbolicExecutor* class returns an instance of the class *SymbolicExecutionTree*. This data structure is depicted in Figure 5.6 and will be described later in the course of this section.

The SMT Solver Yices

As already discussed in Section 4.3.2, the symbolic execution of an IOSTS requires to decide about the feasibility of paths, i.e., the path conditions have to be checked for satisfiability. Furthermore, it is necessary to determine whether some Boolean formula is a tautology in order to identify state inclusions. These two tasks are performed by the SMT solver *Yices*[1] [33], which has won six out of twelve categories in the Satisfiability Modulo Theories Competition of 2007 (SMT-COMP'07[2]). Although a Java API for Yices[3] is available, we have integrated the Yices command line tool. The reasons for this decision are the limited functionality of the Java API as well as problems that were encountered with the underlying library.

[1] `http://yices.csl.sri.com/` (last visit 2009-09-27)
[2] `http://www.smtcomp.org/2007/` (last visit 2009-09-27)
[3] `http://atlantis.seidenberg.pace.edu/wiki/lep/Yices%20Java%20API%20Lite` (last visit 2009-09-27)

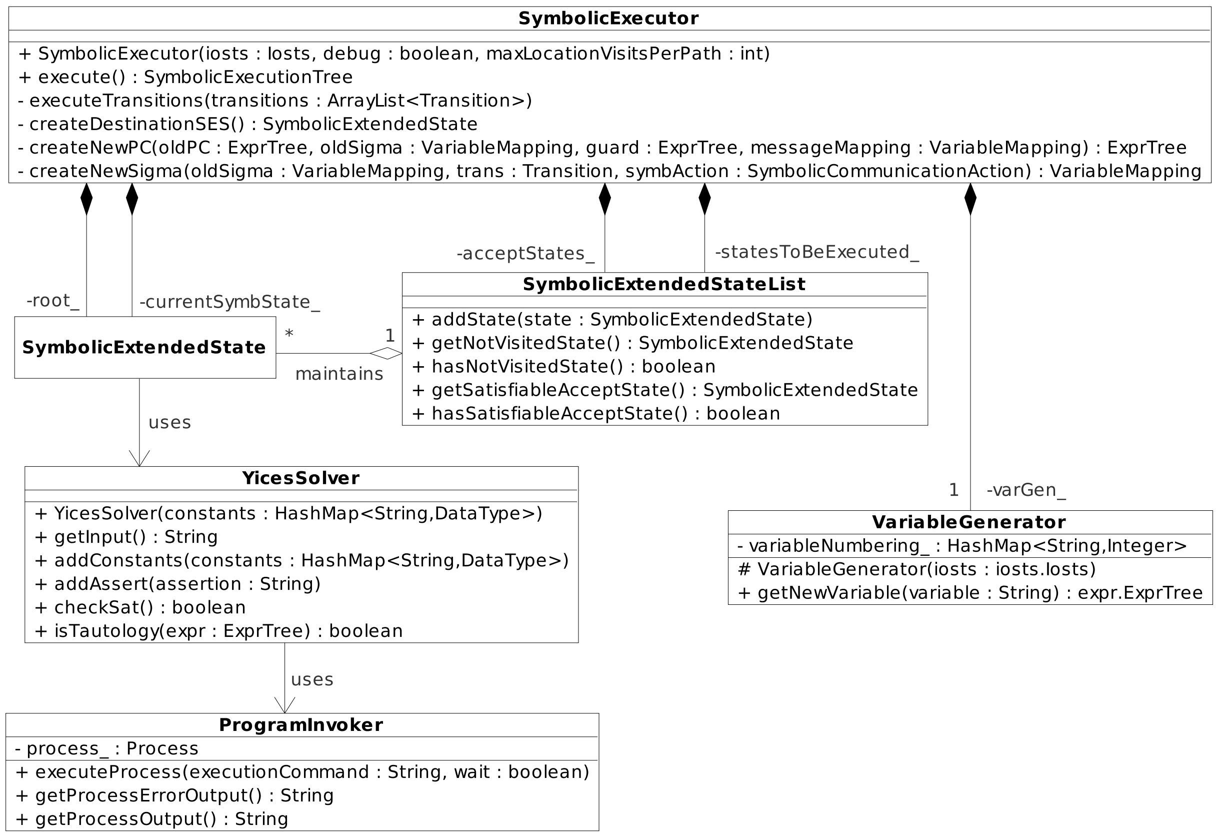

SymbolicExecutor
+ SymbolicExecutor(iosts : Iosts, debug : boolean, maxLocationVisitsPerPath : int)
+ execute() : SymbolicExecutionTree
- executeTransitions(transitions : ArrayList<Transition>)
- createDestinationSES() : SymbolicExtendedState
- createNewPC(oldPC : ExprTree, oldSigma : VariableMapping, guard : ExprTree, messageMapping : VariableMapping) : ExprTree
- createNewSigma(oldSigma : VariableMapping, trans : Transition, symbAction : SymbolicCommunicationAction) : VariableMapping
-root_
-currentSymbState_
-acceptStates_
-statesToBeExecuted_
SymbolicExtendedState
*
1
maintains
SymbolicExtendedStateList
+ addState(state : SymbolicExtendedState)
+ getNotVisitedState() : SymbolicExtendedState
+ hasNotVisitedState() : boolean
+ getSatisfiableAcceptState() : SymbolicExtendedState
+ hasSatisfiableAcceptState() : boolean
uses
YicesSolver
+ YicesSolver(constants : HashMap<String,DataType>)
+ getInput() : String
+ addConstants(constants : HashMap<String,DataType>)
+ addAssert(assertion : String)
+ checkSat() : boolean
+ isTautology(expr : ExprTree) : boolean
1
-varGen_
VariableGenerator
- variableNumbering_ : HashMap<String,Integer>
# VariableGenerator(iosts : iosts.Iosts)
+ getNewVariable(variable : String) : expr.ExprTree
uses
ProgramInvoker
- process_ : Process
+ executeProcess(executionCommand : String, wait : boolean)
+ getProcessErrorOutput() : String
+ getProcessOutput() : String

Figure 5.5: The *SymbolicExecutor* is the central class for the symbolic execution of IOSTS. It implements the algorithm as defined in Section 4.3.2. The *VariableGenerator* is used to generate unique identifiers that serve as symbolic values for the variables and messages. The *YicesSolver* class interfaces with the SMT solver *Yices* and is used to determine the satisfiability of a *SymbolicExtendedState* as well as to decide about state inclusions. The Yices command line tool is started by the *ProgramInvoker* class.

As illustrated in Figure 5.5, the *YicesSolver* class is a wrapper class for the Yices command line tool (version 1.0.21). It is used by the class *SymbolicExtendedState* to calculate its satisfiability and to determine whether it is included in any of the already calculated SES. The *YicesSolver* class calls Yices with the aid of a *ProgramInvoker* class, which starts a new process via Java's *Runtime*.`exec` method. The method takes a *String* object that represents the command that has to be executed and returns an instance of Java's *Process* class, which allows to read its output stream. In this way, the result delivered by Yices can be accessed. The Yices command line tool is invoked by the command "`yices input_file.ys`". Of course, this solution requires that Yices is properly installed. Furthermore, its `bin` folder, which contains the Yices executable, has to be part of the `PATH` system variable.

The input file which is passed to Yices is created by the *YicesSolver* class as a temporary file, which is deleted after the Yices solver has delivered a result. Generally, Yices supports the SMT-LIB standard[4] [65], which defines a common interface for SMT solvers. Hence, by using the SMT-LIB language, it would be easier to integrate a different SMT solver. Nevertheless, we decided to use the specific Yices input language[5], since it is more powerful and flexible [33].

[4]`http://combination.cs.uiowa.edu/smtlib/` (last visit 2009-09-27)
[5]`http://yices.csl.sri.com/language.shtml` (last visit 2009-09-27)

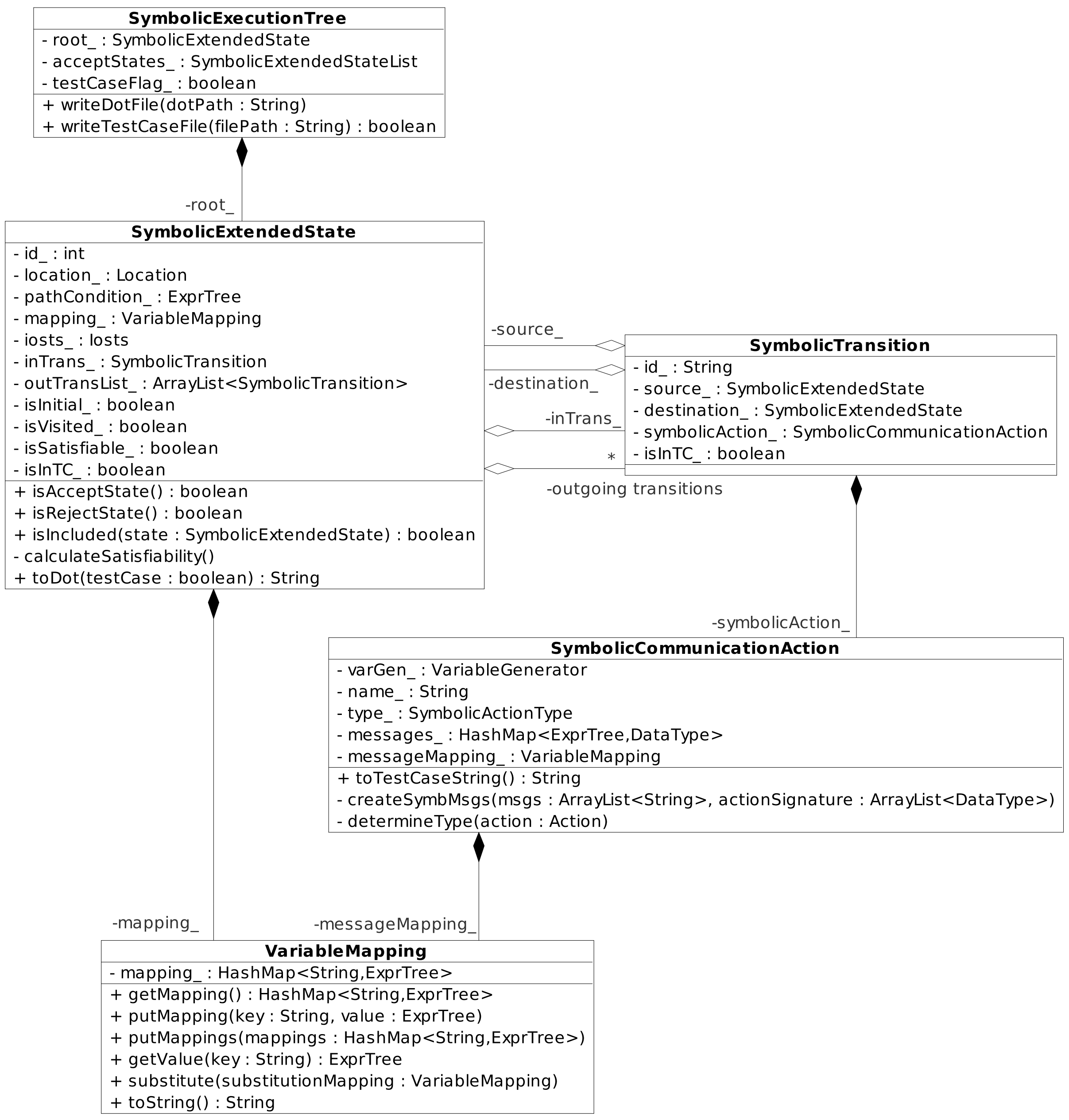

SymbolicExecutionTree
- root_ : SymbolicExtendedState
- acceptStates_ : SymbolicExtendedStateList
- testCaseFlag_ : boolean
+ writeDotFile(dotPath : String)
+ writeTestCaseFile(filePath : String) : boolean
-root_
SymbolicExtendedState
- id_ : int
- location_ : Location
- pathCondition_ : ExprTree
- mapping_ : VariableMapping
- iosts_ : Iosts
- inTrans_ : SymbolicTransition
- outTransList_ : ArrayList<SymbolicTransition>
- isInitial_ : boolean
- isVisited_ : boolean
- isSatisfiable_ : boolean
- isInTC_ : boolean
+ isAcceptState() : boolean
+ isRejectState() : boolean
+ isIncluded(state : SymbolicExtendedState) : boolean
- calculateSatisfiability()
+ toDot(testCase : boolean) : String
-source_
-destination_
-inTrans_
*
-outgoing transitions
SymbolicTransition
- id_ : String
- source_ : SymbolicExtendedState
- destination_ : SymbolicExtendedState
- symbolicAction_ : SymbolicCommunicationAction
- isInTC_ : boolean
-symbolicAction_
SymbolicCommunicationAction
- varGen_ : VariableGenerator
- name_ : String
- type_ : SymbolicActionType
- messages_ : HashMap<ExprTree,DataType>
- messageMapping_ : VariableMapping
+ toTestCaseString() : String
- createSymbMsgs(msgs : ArrayList<String>, actionSignature : ArrayList<DataType>)
- determineType(action : Action)
-mapping_
-messageMapping_
VariableMapping
- mapping_ : HashMap<String,ExprTree>
+ getMapping() : HashMap<String,ExprTree>
+ putMapping(key : String, value : ExprTree)
+ putMappings(mappings : HashMap<String,ExprTree>)
+ getValue(key : String) : ExprTree
+ substitute(substitutionMapping : VariableMapping)
+ toString() : String

Figure 5.6: *SymbolicExecutionTree*s are implemented straight forward with a class for *SymbolicExtendedState*s (SES) and a class for *SymbolicTransition*s. Each SES except for the initial SES has one incoming symbolic transition and zero or more outgoing symbolic transitions. Each symbolic transition has one *SymbolicCommunicationAction*. The message mappings μ_{sa} of the symbolic transitions as well as the variable mappings σ of the symbolic extended states are represented as instances of the class *VariableMapping*.

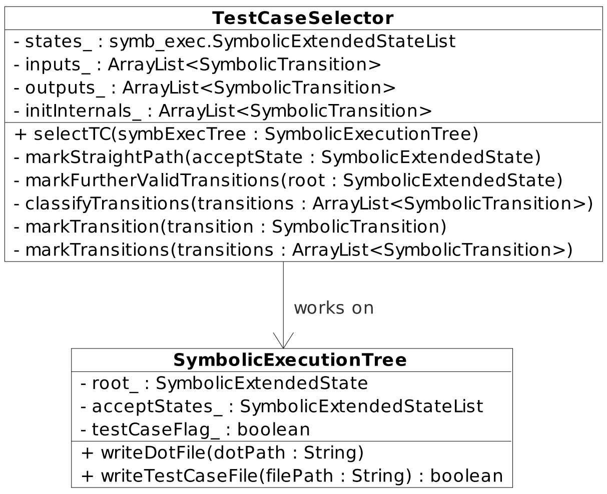

TestCaseSelector
- states_ : symb_exec.SymbolicExtendedStateList
- inputs_ : ArrayList<SymbolicTransition>
- outputs_ : ArrayList<SymbolicTransition>
- initInternals_ : ArrayList<SymbolicTransition>
+ selectTC(symbExecTree : SymbolicExecutionTree)
- markStraightPath(acceptState : SymbolicExtendedState)
- markFurtherValidTransitions(root : SymbolicExtendedState)
- classifyTransitions(transitions : ArrayList<SymbolicTransition>)
- markTransition(transition : SymbolicTransition)
- markTransitions(transitions : ArrayList<SymbolicTransition>)
works on
SymbolicExecutionTree
- root_ : SymbolicExtendedState
- acceptStates_ : SymbolicExtendedStateList
- testCaseFlag_ : boolean
+ writeDotFile(dotPath : String)
+ writeTestCaseFile(filePath : String) : boolean

Figure 5.7: The algorithm for the test case selection from *SymbolicExecutionTree*s is implemented in the class *TestCaseSelector*.

Data Structure for Symbolic Execution Trees

Figure 5.6 shows a class diagram of the data structure used to represent symbolic execution trees. *SymbolicExecutionTree*s are implemented straight forward with a class for *SymbolicExtendedState*s (SES) and a class for *SymbolicTransition*s. Each SES except for the initial SES has one incoming symbolic transition and zero or more outgoing symbolic transitions. Each symbolic transition has one source SES, one destination SES, and one *SymbolicCommunicationAction*. The message mappings μ_{sa} of the symbolic transitions as well as the variable mappings σ of the symbolic extended states are represented as instances of the class *VariableMapping*.

5.2.3 Test Case Selection

The algorithm for the test case selection from symbolic execution trees (see Section 4.4.1) is implemented in the class *TestCaseSelector*. The method `selectTC` takes an instance of a *SymbolicExecutionTree* and selects the straight path from one arbitrarily chosen *Accept* state to the root SES. Subsequently, it investigates further SES according to Step 3 of the selection algorithm. For this purpose, it maintains a list `states_` that corresponds to the set S used in the algorithm. It contains all SES that have to be investigated in the future. Note that no new instance of the *SymbolicExecutionTree* is created. The *TestCaseSelector* just sets the `isInTC_` flags of the *SymbolicExtendedState*s in the symbolic execution tree. Avoiding the creation of a new *SymbolicExecutionTree* object saves computation time and memory.

5.2.4 Program Logic

Figure 5.8 shows the central classes of our Java implementation. The controller class *SymbTCGen* (Symbolic Test Case Generator) contains the `main` method of our application. It parses the command line arguments and implements the test case generation process by calling the *IostsBuilder* with the `dot` and `pdf` files, which were specified as command line arguments. Subsequently, it passes the generated *Iosts* to an instance of the *SymbolicExecutor* and calls its `execute` method. Finally, an instance of the *TestCaseSelector* is invoked with the resulting *SymbolicExecutionTree*.

Additionally to the test case generation process, the *SymbTCGen* class is responsible for exporting the parsed *Iosts* and the *SymbolicExecutionTree* into `dot` and `pdf` format if the command line flag `debug` is set. The resulting test case, which is represented by an instance of the class *SymbolicExecutionTree*, is transformed into `dot`, `pdf`, and a text format for interfacing with a new test execution

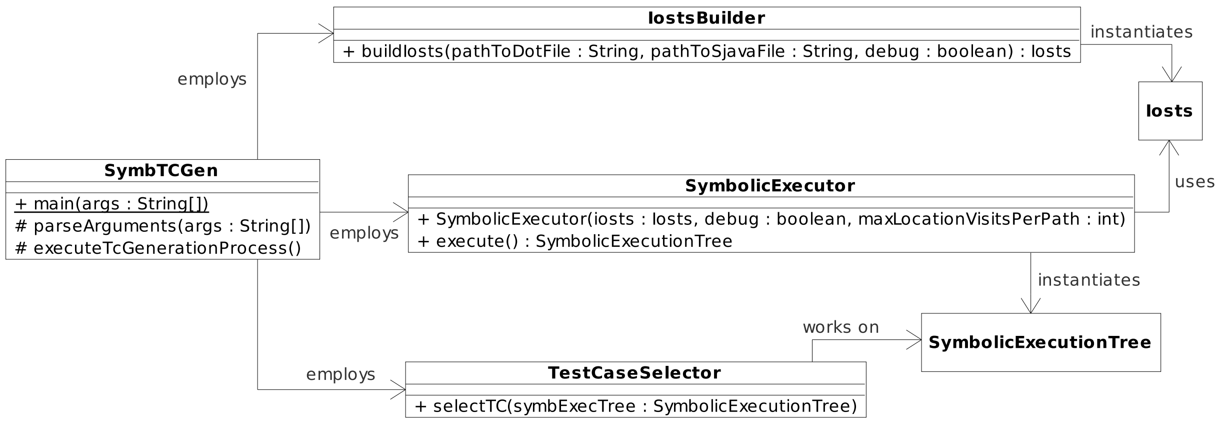

IostsBuilder
+ buildIosts(pathToDotFile : String, pathToSjavaFile : String, debug : boolean) : Iosts
instantiates
Iosts
employs
SymbTCGen
+ main(args : String[])
# parseArguments(args : String[])
# executeTcGenerationProcess()
employs
SymbolicExecutor
+ SymbolicExecutor(iosts : Iosts, debug : boolean, maxLocationVisitsPerPath : int)
+ execute() : SymbolicExecutionTree
uses
instantiates
works on
SymbolicExecutionTree
employs
TestCaseSelector
+ selectTC(symbExecTree : SymbolicExecutionTree)

Figure 5.8: The class *SymbTCGen* contains the `main` method of our application. It parses the command line arguments and implements the test case generation process by invoking the *IostsParser*, the *SymbolicExecutor*, and the *TestCaseSelector*.

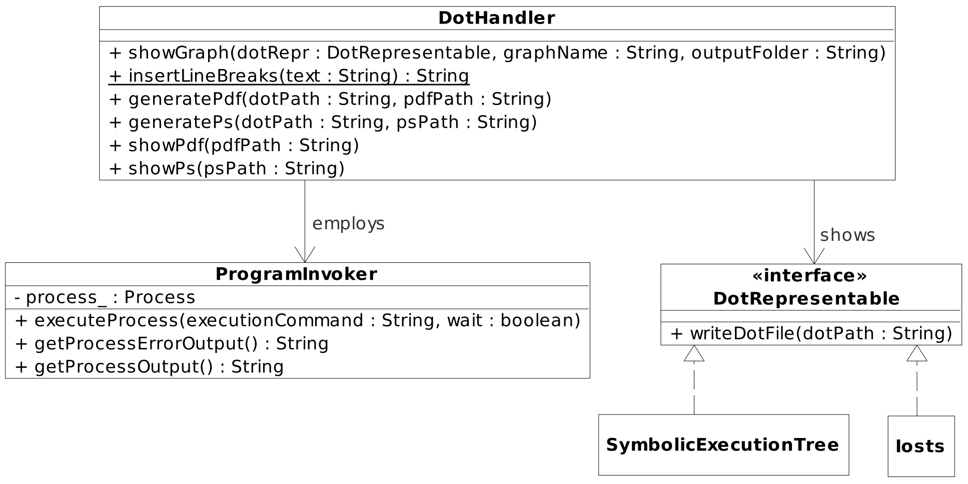

DotHandler
+ showGraph(dotRepr : DotRepresentable, graphName : String, outputFolder : String)
+ insertLineBreaks(text : String) : String
+ generatePdf(dotPath : String, pdfPath : String)
+ generatePs(dotPath : String, psPath : String)
+ showPdf(pdfPath : String)
+ showPs(psPath : String)
employs
shows
ProgramInvoker
- process_ : Process
+ executeProcess(executionCommand : String, wait : boolean)
+ getProcessErrorOutput() : String
+ getProcessOutput() : String
«interface»
DotRepresentable
+ writeDotFile(dotPath : String)
SymbolicExecutionTree
Iosts

Figure 5.9: The class *DotHandler* generates `dot` files for classes that implement the interface *DotRepresentable*. Additionally, it supports the conversion of `dot` files into `pdf` or `ps` format and is able to display such files.

framework. For this purposes, it uses the method `showGraph` of the *DotHandler* class, which gets the `dot` representation of the *Iosts* and the *SymbolicExecutionTree* instances and converts them into `pdf` or `ps` (Post Script) format with the `dot` command provided by the Graphviz tool set[6]. Furthermore, it allows to open the generated `pdf` file with the tool *Xpdf*[7] and the generated `ps` file with the Ghostview implementation *gv*[8]. Figure 5.9 shows the classes that are involved in this process.

5.2.5 External Dependencies

As already mentioned in the previous sections, our Java implementation depends on several external tools. Since most of them are only available for Linux platforms, our implementation does not support other operating systems either. The external dependencies of our Java application are:

[6] `http://www.graphviz.org/` (last visit 2009-09-27)
[7] `http://www.foolabs.com/xpdf/` (last visit 2009-09-27)
[8] `http://pages.cs.wisc.edu/~ghost/gv/index.htm` (last visit 2009-09-27)

- The Yices SMT solver[9]: command line tool, version 1.0.21
- Graphviz[10]: graph visualization software
- Xpdf[11]: a PDF viewer for X
- gv[12]: implementation of Ghostview, allows to view and navigate PostScript files
- ANTLR Parser Generator[13]: `jar` files, version 3.0

5.2.6 Usage

Our Java application is started with the following command:

```
main.SymbTCGen <path to input dot file>
               <path to input sjava file>
               <path for output file storage>
               <debug flag>
               <upper bound>
```

Note that the ordering of the command line arguments is relevant and that all five arguments have to specified. They have the following meaning:

path to input dot/sjava file The first two arguments specify the path to the `dot` file and `sjava` file respectively. They contain the closed product of the specification and a test purpose in the form of an IOSTS, which will be used to calculate a test case. Both files can be automatically generated with STG.

path for output file storage The third command line argument specifies where the generated output files shall be stored.

debug flag The fourth argument can be "*true*" or "*false*". If the debug flag is set, additional debug information will be printed and intermediate results, i.e., the parsed IOSTS and the generated symbolic execution tree, will be exported to `dot` and `pdf` format.

upper bound The last argument is an integer value greater than zero. It specifies an upper bound for loop unfolding, i.e., the maximum number of visits of one IOSTS location in each path of the symbolic execution tree.

The following entries have to be specified in Java's classpath:

- ANTLR:
  - antlr-2.7.7.jar
  - antlr-3.0.jar
  - antlr-runtime-3.0.jar
  - stringtemplate-3.0.jar
- `bin` folder containing the binaries of our Java implementation

[9]`http://yices.csl.sri.com/` (last visit 2009-09-27)
[10]`http://www.graphviz.org/` (last visit 2009-09-27)
[11]`http://www.foolabs.com/xpdf/` (last visit 2009-09-27)
[12]`http://pages.cs.wisc.edu/~ghost/gv/index.htm` (last visit 2009-09-27)
[13]`http://www.antlr.org/` (last visit 2009-09-27)

5.3 Shell Script Combining STG and the Java Implementation

As already mentioned in Section 5.1, STG and our new Java implementation have been integrated into one application by the use of a shell script (see Listing 5.1). The script was written for the *Bash* shell and expects five arguments:

1. The first argument specifies the name of the STG file, which contains the specification of the SUT and the test purpose, without any path information. The file is supposed to be in a folder named `spec`, which has to be located in the same directory as the script.
2. The name of the IOSTS that represents the system specification.
3. The name of the IOSTS that represents the test purpose.
4. The debug flag is directly passed to the Java program (see Section 5.2.6).
5. The last argument is supposed to be an integer value and will be used as the upper bound for the Java program (see Section 5.2.6).

In the lines 2 to 6 of Listing 5.1, the script prints information about its usage if the number of arguments with which it was called is wrong. The lines 9 to 13 carry out preparations that are necessary in order to execute STG (see Section 3.3.2).

Since we do not need to execute STG's complete test case generation process, we use a command file to specify the operations that have to be carried out by STG. The shell script requires a template command file called `cmdfile` to be in the same directory as the script itself. Its content must be the same as the STG commands presented in Listing 5.2. The lines 16 to 19 of the shell script modify the command file by inserting the name of the specification (second argument of the script) and the name of the test purpose (third argument of the script).

Line 22 changes into the directory `spec`, which contains the STG file. The NBac library files, which are located in `stg_linux/lib`, are copied directly into the `spec` directory (lines 25 and 26). This is necessary, because STG expects the NBac library to be in the same folder as the file it is processing.

Line 31 invokes STG, which calculates the closed product between the specification and the test purpose according to the operations specified in the command file. Listing 5.2 shows its content: At first, the specification (Line 1) and the test purpose (Line 2) are parsed. Subsequently, the test purpose is completed (Line 3) and the product between the specification and the completed test purpose is calculated (Line 4). Finally, the product is closed (Line 5) and exported into `dot` (Line 6) and `sjava` format (Line 7).

After STG has finished, the script shown in Listing 5.1 invokes our Java program (Line 45). The classpath is set according to Section 5.2.6. The arguments passed to the program are:

1. The `dot` file generated by STG: Its name consists of the name of the STG file without file extension concatenated with the string "_closure.dot".
2. The `sjava` file generated by STG: Its name consists of the name of the STG file without file extension concatenated with the string "Test.sjava".
3. The third argument specifies the folder in which outputs will be stored. The script uses the current folder (`spec`).
4. The debug flag is specified by the user as the fourth argument of the script.
5. The upper bound is specified by the user as the last argument of the script.

This completes the presentation of our tool. The next chapter reports about its application in three case studies. In our discussion about future work (see Section 8.3), we will make some suggestions about how this prototype implementation can be enhanced.

```
# check arguments
if [ $# −ne 5 ]; then
  echo Wrong number of parameters!
  echo Usage: $0 ”<file.stg> <test name> <test purpose name> <debug> <upper bound>”
  exit 127
fi

# set PATH
export STG_LINUX=/stg_linux
export PATH=$PATH:$STG_LINUX

# copy ”rung” file in order to be able to execute the stg tool
cp $STG_LINUX/tmp/rung /tmp/rung

# put correct test name and test purpose name into command file
touch cmdfile.tmp
sed s/”spec := getiosts(”[.]*”)”/”spec := getiosts(”$2”)”/ <cmdfile > cmdfile.tmp
sed s/”tp := getiosts(”[.]*”)”/”tp := getiosts(”$3”)”/ <cmdfile.tmp > cmdfile
rm cmdfile.tmp

# go to directory ”spec” (contains <file.stg>)
cd spec

# create folder ”lib” in working directory and copy NBAC library into it
mkdir −p ./lib/nbac
cp $STG_LINUX/lib/nbac/* ./lib/nbac

# generate closed product with STG
# stg <file>.stg −cmdfile cmdfile
echo ”\n>>> Starting STG...\n”
stg $1 −cmdfile ../cmdfile

# set path variables for ANTLR and our Java implementation
ANTLR_LIB=/antlr−3.0/lib
SYMB_TC_GEN=/thesis/implementation/bin

# cut file extension from STG file and build names for DOT and SJAVA file
BASENAME=‘basename $1 .stg‘
DOT=”${BASENAME}_closure.dot”
SJAVA=”${BASENAME}Test.sjava”

# invoke Java implementation SymbTCGen
# main.SymbTCGen <path to input dot file> <path to input sjava file> <path for output file storage> <debug_flag = true || false> <upper bound>
echo ”\n>>> Starting SymbTCGen...\n”
java −classpath ”$SYMB_TC_GEN:$ANTLR_LIB/antlr−2.7.7.jar:$ANTLR_LIB/antlr−3.0.jar:$ANTLR_LIB/antlr−runtime−3.0.jar:$ANTLR_LIB/stringtemplate−3.0.jar” main.SymbTCGen $DOT $SJAVA . $4 $5
```

Listing 5.1: Shell script for integrating STG and our new Java implementation into one application.

```
spec := getiosts(...);
tp := getiosts(...);
complete_tp := complete(tp);
product := spec*complete_tp;
closed_p := close(product);
show closed_p;
tojava closed_p
```

Listing 5.2: Command file template for calculating the closed product with STG.

6 Case Studies and Results

This chapter shows the applicability of the developed test generation tool. Since a full presentation of the used specifications and test purposes as well as of the resulting symbolic execution trees and test cases would be too extensive, this chapter will only give metrics about them. Furthermore, the performance in terms of time of STG's test case generation process will be compared to the performance of our approach. For our implementation, the elapsed time was measured from executions without having the `debug` flag set, i.e., no debug output was printed and the parsed IOSTS as well as the symbolic execution tree were not exported to `dot` and `pdf` format respectively. All experiments were run on the following system:

- Intel® Core™2 Duo Processor T7200 (2.00 GHz)
- 2 GB RAM
- Ubuntu 8.04
- Java Runtime Environment: java version 1.6.0_0
- Java Compiler: javac 1.6.0_0-internal
- STG: unversioned, downloaded from the STG web page[1] dated with *August 28, 2008*
- Yices SMT solver: command line tool, version 10.0.21
- *Bash* shell: version 3.2.39(1)

The case studies used to evaluate our test case generation approach consist of three examples of different size. The first one is called Triangle Type Checker and is a very small example, which has already been used for illustration throughout this work. The second case study is of industrial relevance. It concerns a Session Initiation Protocol (SIP) Registrar, which is informally specified in RFC 3261. The Conference Protocol is the subject of the third case study. It has already served as a case study during the development of other formal testing tools. Note that the case studies cover test case generation and do not deal with test case execution.

6.1 Triangle Type Checker

This section deals with the Triangle Type Checker example, which has already been introduced in Section 2.2.2. The original idea for this example stems from Myers [58]. The Triangle Type Checker takes three integer values as inputs. They are representing the three side lengths of a triangle. Afterwards, the Triangle Type Checker determines whether these three side lengths form a valid triangle. If they do, the type of the triangle (equilateral, isosceles, or scalene) is decided. If one of the side lengths is negative or zero, "NotPositive" is output. If the three side lengths do not form a valid triangle, "NotTriangle" is reported.

6.1.1 Specification

The IOSTS representing the specification of this example has already been shown in Figure 2.2. The Triangle Type Checker example is a small example that is used for demonstration throughout the STG website[2]. Table 6.1 describes the size of the Triangle Type Checker specification, which consists of about 80 lines of code. It needs only 3 variables for its data and has only 6 locations connected via 9 transitions with simple guards. The biggest guard contains just 10 operators from which 5 are logical operators. Almost all actions of this specification do not carry messages.

[1] `http://www.irisa.fr/prive/ployette/stg-doc/stg-web_5.html` (last visit 2009-09-27)
[2] `http://www.irisa.fr/prive/ployette/stg-doc/stg-web.html` (last visit 2009-09-27)

Specification (Triangle Type Checker)		
Category	**Metric**	**Value**
general information	lines of code	~ 80
	# system parameters	0
	# variables	3
	# locations	6
	# transitions	9
actions	# different input actions	1
	# different output actions	6
	# different internal actions	1
# messages of different actions	minimum	0
	maximum	3
	average	0
# operators in guards (logical, arithmetic, and comparison operators)	minimum	0
	maximum	10
	average	5
# logical operators in guards (not, and, or)	minimum	0
	maximum	5
	average	2

Table 6.1: This table describes the size of the Triangle Type Checker specification.

6.1.2 Test Case Generation

Test Purposes

Six test purposes, which have already been designed by the STG team, were used for this case study. They can be found in the file `triangle.stg`, which is available at the STG web page[3]. Table 6.2 gives an overview of the size of the (intermediate) results produced throughout test case generation. The size of the different test purposes is presented at the top of the table. Since the specification itself is quite small, the test purposes are very simple. Each of them consists of approximately 20 lines of code, 3 locations, and 2 transitions.

Closed Products

The test case generation starts with the calculation of the products between the specification and a test purpose. Subsequently, these products are closed. These *closed products*, which are calculated by STG, are the input for the new test case generation approach of this work (see Chapter 4). In Table 6.2, the size of the closed products for the Triangle Type Checker example is described by five metrics: (1) the number of locations, (2) the number of transitions, (3) the number of system parameters, (4) the number of variables, and (5) the number of different message names. Since the used test purposes have the same structure, the closed products are of the same size. Each of them consists of 7 locations, which are connected via 10 transitions. None of the closed products has any system parameters. Each of them has 3 variables and 3 different message names.

[3]`http://www.irisa.fr/prive/ployette/stg-doc/triangle.stg` (last visit 2009-09-27)

Test Case Generation (Triangle Type Checker)								
		TP_NotPositive	**TP_NotTriangle**	**TP_IsTriangle**	**TP_Equilateral**	**TP_Isosceles**	**TP_Scalene**	**Average**
Test Purposes	lines of code	∼ 20	∼ 20	∼ 20	∼ 20	∼ 20	∼ 20	∼ 20
	# locations	3	3	3	3	3	3	3
	# transitions	2	2	2	2	2	2	2
Closed Products	# locations	7	7	7	7	7	7	7
	# transitions	10	10	10	10	10	10	10
	# system parameters	0	0	0	0	0	0	0
	# variables	3	3	3	3	3	3	3
	# messages	3	3	3	3	3	3	3
Symbolic Execution Trees	# SES	9	9	6	9	9	9	9
	# satisfiable *Accept* states	1	1	1	1	1	1	1
	upper bound	2	2	2	2	2	2	2
Generated Test Cases	# SES	9	9	6	9	9	9	9
	# SES in longest path	5	5	4	5	5	5	5
	# SES in shortest path	4	4	4	4	4	4	4
	# Pass	1	1	1	1	1	1	1
Test Case Generation Time [sec]	STG	2.14	1.99	2.25	1.68	3.51	3.85	2.57
	regular termination	yes	yes	yes	yes	yes	yes	–
	New Approach	2.17	1.05	1.22	0.91	0.95	1.03	1.22
	STG	0.09	0.17	0.15	0.21	0.14	0.23	0.17
	own implementation	2.08	0.88	1.07	0.7	0.81	0.8	1.05
	regular termination	yes	yes	yes	yes	yes	yes	–

Table 6.2: This table describes the process of generating test cases for the Triangle Type Checker. It states the size of the intermediate and final results as well as the time needed to generate test cases for different test purposes.

Symbolic Execution Trees

Next, the closed products are symbolically executed in order to generate test cases. Some metrics describing the resulting symbolic execution trees are presented in the middle part of Table 6.2. The symbolic execution tree for the test purpose "TP_IsTriangle" has 6 Symbolic Extended States (SES). Each of the other symbolic execution trees consists of 9 SES. None of the symbolic execution trees contains more than one satisfiable *Accept* state. Hence, at most one test case per test purpose can be generated at all. The upper bound for loop unfolding was chosen to be 2, i.e., each path of the symbolic execution tree has at most 2 SES corresponding to the same IOSTS location (see Section 4.3.2 for more details). For the small Triangle Type Checker example, in which neither the specification nor any of the test purposes contains any loops, an upper bound of value 2 is sufficient.

Note that all symbolic execution trees except for the one for the test purpose "TP_IsTriangle" have the same structure, but they are not exactly the same. The symbolic execution tree for "TP_IsTriangle" is smaller, because it reaches an *Accept* state before deciding about the type of the triangle. Hence, three SES (for the three triangle types) are not part of this symbolic execution tree.

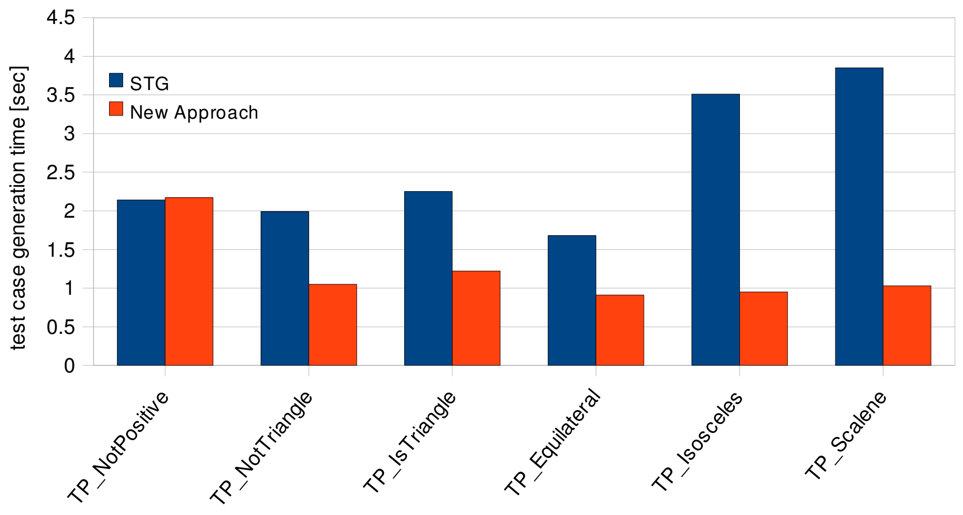


Figure 6.1: This diagram illustrates the time consumed by STG compared to the time needed by our approach to generate test cases for the Triangle Type Checker.

Generated Test Cases

After the symbolic execution of the closed products, symbolic test cases are extracted from the symbolic execution trees according to the test case selection algorithm defined in Section 4.4.1. Table 6.2 describes the resulting test cases with the aid of four metrics: (1) the total number of SES in each test case, (2) the number of SES in the longest path, (3) the number of SES in the shortest path, and (4) the number of *Pass* verdicts in each test case. The test cases generated for the Triangle Type Checker comprise the whole corresponding symbolic execution trees, i.e., the symbolic execution trees are equivalent to the generated test cases. The shortest path in each test case consists of 4 SES. The longest path comprises 5 SES for all test cases except of "TP_IsTriangle", which has only 4 SES in its longest path. Each test case contains one *Pass* verdict, i.e., one *Accept* state.

Test Case Generation Time

At the bottom of Table 6.2, the amount of time that was needed for test case generation by STG and the amount of time that was taken by our approach are listed. Additionally, the elapsed time for test case generation with our approach is broken down into two parts: (1) the amount of time that is needed by STG to complete the test purpose, to calculate the product, and to close the product, and (2) the amount of time that is consumed by the newly implemented parts of our approach, i.e., by the symbolic execution of the closed product and the test case selection from the resulting symbolic execution tree. Note that all values are given in seconds and were measured from program executions without printing debug output and without exporting intermediate results like the parsed IOSTS as well as the generated symbolic execution tree into `dot`/`pdf` format. The rows labelled by *regular termination* state whether STG and the program which implements our approach terminated successfully, i.e., whether they generated test cases.

Both, STG as well as our program, are able to generate test cases for each of the test purposes for the Triangle Type Checker. At an average, our implementation is faster than STG. Just for one test purpose, which is "TP_NotPositive", STG is slightly faster than our approach. Figure 6.1 illustrates the difference between the time consumed by STG and the time needed by our approach. STG's proportion of time for generating test cases with our approach is rather low. On average, the time needed by the reused components of STG forms about 13.8 % of the overall test case generation time of our approach.

Example Test Case

Since a full presentation of all generated test cases would be too extensive, we give just one example of a test case for the Triangle Type Checker. The test case for the test purpose “TP_NotTriangle” (depicted in Figure 2.3) tests whether the IUT correctly identifies invalid triangles. The generated test case has already been shown in Figure 4.4.

6.2 Session Initiation Protocol

The Session Initiation Protocol (SIP) is of industrial relevance. It is a signalling protocol for communication sessions between two end points and is informally specified in RFC 3261 [66]. Since SIP is independent of the exchanged media type, it can be used for various applications, e.g., internet telephony (VoIP), multimedia distribution, or multimedia conferences. There exist many implementations of the Session Initiation Protocol, e.g., OpenSIPS (Open SIP Server)[4] or Asterisk[5], which can be tested with the test cases generated by our approach.

According to Aichernig et al. [2], the functionalities provided by SIP can be divided into two categories: session management and user management. Session management comprises the establishment, transfer, and termination of sessions as well as the modification of session parameters. User management includes functionalities like the determination of the user location and of the user availability. The ability of SIP to locate users allows them to have just one externally visible identifier, although they may access the SIP network from different network locations. The therefore necessary location information about users is maintained by the so-called *SIP Registrar*, which is the focus of this case study.

SIP is a text based protocol that works with the concept of requests and responses. The basic version of SIP knows six different request methods. The request method that is primarily used in connection with the SIP Registrar is called `REGISTER` and associates a user address with an end point. Requests as well as responses are transferred in the form of messages. Each message has the same structure. It consists of a start line, a message header, and a message body. The start line indicates the request method or response type. The message header gives information about the message, e.g., the sender, the receiver, or the content type. For example, the message header of the `REGISTER` request method may contain `CONTACT` header fields that are used to modify already stored user location information. The message body may contain data. For the `REGISTER` request method, it is usually empty [2].

Aichernig et al. [2] used the formal testing tool TGV for conformance testing of a SIP Registrar. The used LOTOS specification was developed by Weiglhofer [78]. This work also concentrates on test case generation for the SIP Registrar.

6.2.1 Specification

Table 6.3 describes the size of the specification of the SIP Registrar, which is the biggest of the three examples used for this case study. It consists of 530 lines of STG code in which 2 system parameters and 28 variables are defined. The IOSTS consists of 24 locations and 52 transitions. The protocol specification uses three different actions: one input action, one output action, and one internal action. The minimum number of messages for one action is 5, the maximum is 10. At an average, each used action carries 8 messages. The guards of some transitions are rather large. The maximum number of operators in one guard is 66. The maximum number of logical operators in one guard is 35. At an average, each guard consists of 14 operators from which 7 are logical operators. However, some transitions are not guarded at all.

[4]`http://www.opensips.org/` (last visit 2009-09-27)
[5]`http://www.asterisk.org` (last visit 2009-09-27)

Specification (SIP Registrar)		
Category	**Metric**	**Value**
general information	lines of code	~ 530
	# system parameters	2
	# variables	28
	# locations	24
	# transitions	52
actions	# different input actions	1
	# different output actions	1
	# different internal actions	1
# messages of different actions	minimum	5
	maximum	10
	average	8
# operators in guards (logical, arithmetic, and comparison operators)	minimum	0
	maximum	66
	average	14
# logical operators in guards (not, and, or)	minimum	0
	maximum	35
	average	7

Table 6.3: This table describes the size of the SIP Registrar specification.

6.2.2 Test Case Generation

Test Purposes

Table 6.4 gives an overview of the size of the (intermediate) results produced throughout test case generation. The different test purposes are described at the top of the table. Three metrics are used to describe their size: (1) the lines of code needed to specify them in STG format, (2) the number of locations, and (3) the number of transitions that each test purposes contains. The used test purposes are very different in their size. Two of them are specified in about 90 lines of STG code and consist of 7 locations and 17 to 18 transitions. The other test purposes are significantly smaller. Three of them are written in 30 to 35 lines of code and contain only 4 locations and 5 to 6 transitions. One test purpose comprises 40 lines of STG code and contains 5 locations and 5 transitions. An average test purpose for the SIP Registrar comprises 53 lines of STG code and consists of 5 locations connected via 9 transitions.

Closed Products

The second part of Table 6.4 describes the closed products generated by STG. Five metrics state the size of each closed product. They count the number of (1) locations, (2) transitions, (3) system parameters, (4) variables, and (5) different message names. Note that the values given for the number of transitions do not include duplicates, since redundant transitions are removed from the closed products before they are symbolically executed. At an average, each closed product consists of 65 locations connected via 306 transitions. Each of them has 2 system parameters, 28 variables, and 11 different message names.

Symbolic Execution Trees

The symbolic execution trees for the closed products are described in the middle of Table 6.4. At an average, each symbolic execution tree contains 210 symbolic extended states (SES), whereof 28 are

Test Case Generation (SIP Registrar)								
		register_delete_tp	**register_invalidrequest_tp**	**register_notfound_tp**	**register_ok_tp**	**register_unauthorized_tp**	**register_notfound_guarded_tp**	**Average**
Test Purposes	lines of code	∼ 90	∼ 30	∼ 30	∼ 35	∼ 90	∼ 40	∼ 53
	# locations	7	4	4	4	7	5	5
	# transitions	18	5	5	6	17	5	9
Closed Products	# locations	112	30	30	30	113	73	65
	# transitions	562	183	183	217	482	211	306
	# system parameters	2	2	2	2	2	2	2
	# variables	28	28	28	28	28	28	28
	# messages	11	11	11	11	11	11	11
Symbolic Execution Trees	# SES	566	101	101	125	328	38	210
	# satisfiable *Accept* states	100	18	4	8	38	1	28
	upper bound	2	2	2	2	2	2	2
Generated Test Cases	# SES	38	15	15	17	29	6	20
	# SES in longest path	7	4	4	4	6	4	5
	# SES in shortest path	4	4	4	4	4	4	4
	# Pass	24	9	2	4	9	1	8
Test Case Generation Time [sec]	STG	30 min	13 min	18 min	16 min	2 min	4 min	14 min
	regular termination	no	no	no	no	no	no	–
	New Approach	14.35	4.66	4.43	4.56	11.89	3.93	7.31
	STG	3.16	1.33	1.33	1.78	3.37	1.31	2.05
	own implementation	11.19	3.33	3.1	2.78	8.52	2.62	5.26
	regular termination	yes	yes	yes	yes	yes	yes	–

Table 6.4: This table describes the process of generating test cases for the SIP Registrar. It states the size of the different intermediate and final results as well as the time needed to generate test cases for different test purposes.

satisfiable *Accept* states. Figure 6.2 shows that except for two test purposes, the single values show a correlation between the size of the test purposes (measured in the number of locations and transitions) and the size of the resulting symbolic execution trees (measured in the number of SES). The upper bound used for symbolic execution has been 2 for all test purposes, i.e., each path of the symbolic execution tree has at most 2 SES corresponding to the same IOSTS location (see Section 4.3.2 for more details).

Generated Test Cases

Table 6.4 also describes the test cases that were extracted from the symbolic execution trees according to the algorithm defined in Section 4.4.1. Therefor, four metrics are used: (1) the total number of SES

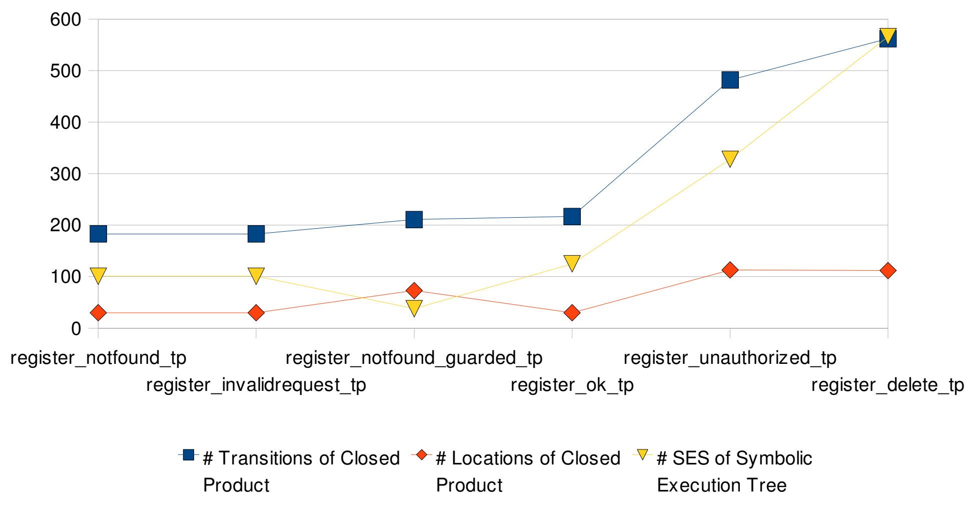


Figure 6.2: For the SIP Registrar example, the size of the closed products show a correlation with the size of the resulting symbolic execution trees except for two test purposes.

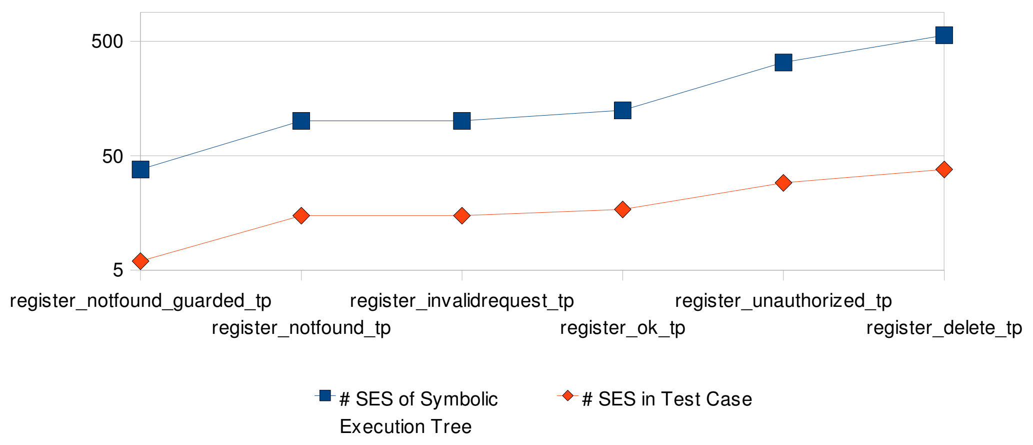


Figure 6.3: The size of the symbolic execution trees and the size of the generated test cases in terms of their number of SES correlate.

in each test case, (2) the number of SES in the longest path, (3) the number of SES in the shortest path, and (4) the number of *Pass* verdicts in each test case. At an average, each test case consists of 20 SES, whereof 8 are *Accept* states, i.e., *Pass* verdicts. The average number of SES in the shortest paths of the symbolic execution trees is 4. The longest paths consist of 5 SES on average. As can be seen in Figure 6.3, the single values show a correlation between the size of the symbolic execution trees and the size of the generated test cases.

Test Case Generation Time

Finally, the amount of time that was needed for test case generation by STG and the amount of time that was taken by our approach are listed at the bottom of Table 6.4. The elapsed time for test case generation with our approach is broken down into two parts: (1) the amount of time that is needed by STG to complete the test purpose, to calculate the product, and to close the product, and (2) the amount of time that is consumed by the newly implemented parts of our approach, i.e., by the symbolic execution of the closed product and the test case selection from the resulting symbolic execution tree. Note that all values which are not listed with a specifically unit are given in seconds. All values were measured from program executions without printing debug output and without exporting intermediate results, i.e, the

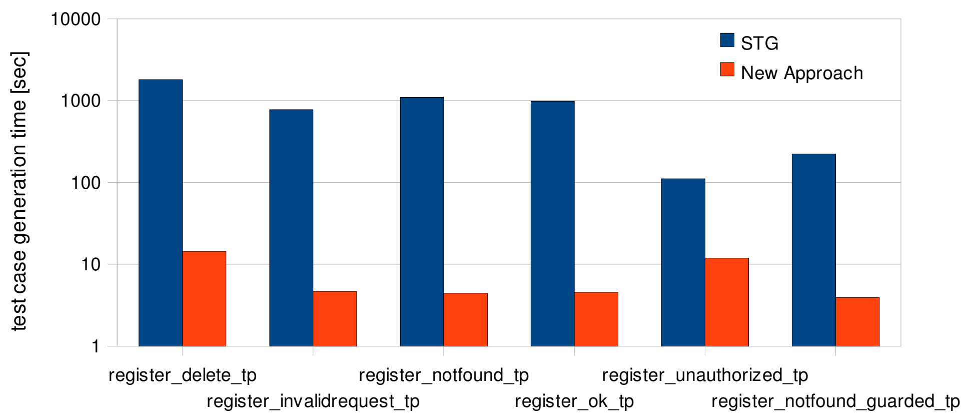


Figure 6.4: This diagram illustrates the time consumed by STG compared to the time needed by our approach to generate test cases for the SIP Registrar. Note that STG failed to generate test cases for all of the used test purposes. The given time values state after which period of time an error occurred.

parsed IOSTS as well as the generated symbolic execution tree, into `dot`/`pdf` format. The rows labelled by *regular termination* state whether STG and the program which implements our approach terminated successfully, i.e., whether they generated test cases.

For the SIP Registrar specification and its 6 different test purposes, STG was not able to generate any test cases. The test case generation for the test purposes “register_invalidrequest_tp”, “register_notfound_tp”, “and register_ok_tp” were aborted with the following error message:

```
Fatal error:  exception Failure("Buffer.add:  cannot grow buffer")
```

The test purposes “register_unauthorized_tp” and “register_notfound_guarded_tp” caused a stack overflow. STG’s error message was:

```
Fatal error:  exception Stack_overflow
```

For the test purpose “register_delete_tp”, STG exceeded the limit of 2.5 GB of RAM.

By contrast, our approach successfully generated test cases for all of the used test purposes within seconds. Figure 6.4 illustrates the elapsed time until STG reported an error and the time needed by our approach to generate test cases. For none of the used test purposes, our approach took longer than 15 seconds. The average time needed for test case generation was about 7 seconds.

Example Test Case

Figure 6.5 serves as an example of a generated test case. The symbolic extended states and symbolic transitions of the test case have the same structure as already explained in Section 4.3.3. For this example, the path conditions and symbolic values for each SES as well as the symbolic action messages and message mappings for each symbolic transition have been removed in order to present the test case on a single A4 page.

The depicted test case corresponds to the test purpose “register_notfound_tp”. It tests whether the IUT correctly implements the following behaviour: If the SIP Registrar receives a `REGISTER` request for a user for which it is not responsible, it has to answer it with the message “`404 (Not Found)`” and must stop the registration process.

Test cases generated with our tool contain implicit verdicts, which have to be made explicit during test case execution. If the test execution ends in an *Accept* state, which is specified through the test purpose, then the verdict is *Pass*. The test case depicted in Figure 6.5 contains nine *Pass* verdicts represented by the nine SES labelled with “Comp_Accept” in the last row of the tree.

All other leaves in the tree structure represent the verdict *Inconclusive*, since no *Accept* state can be

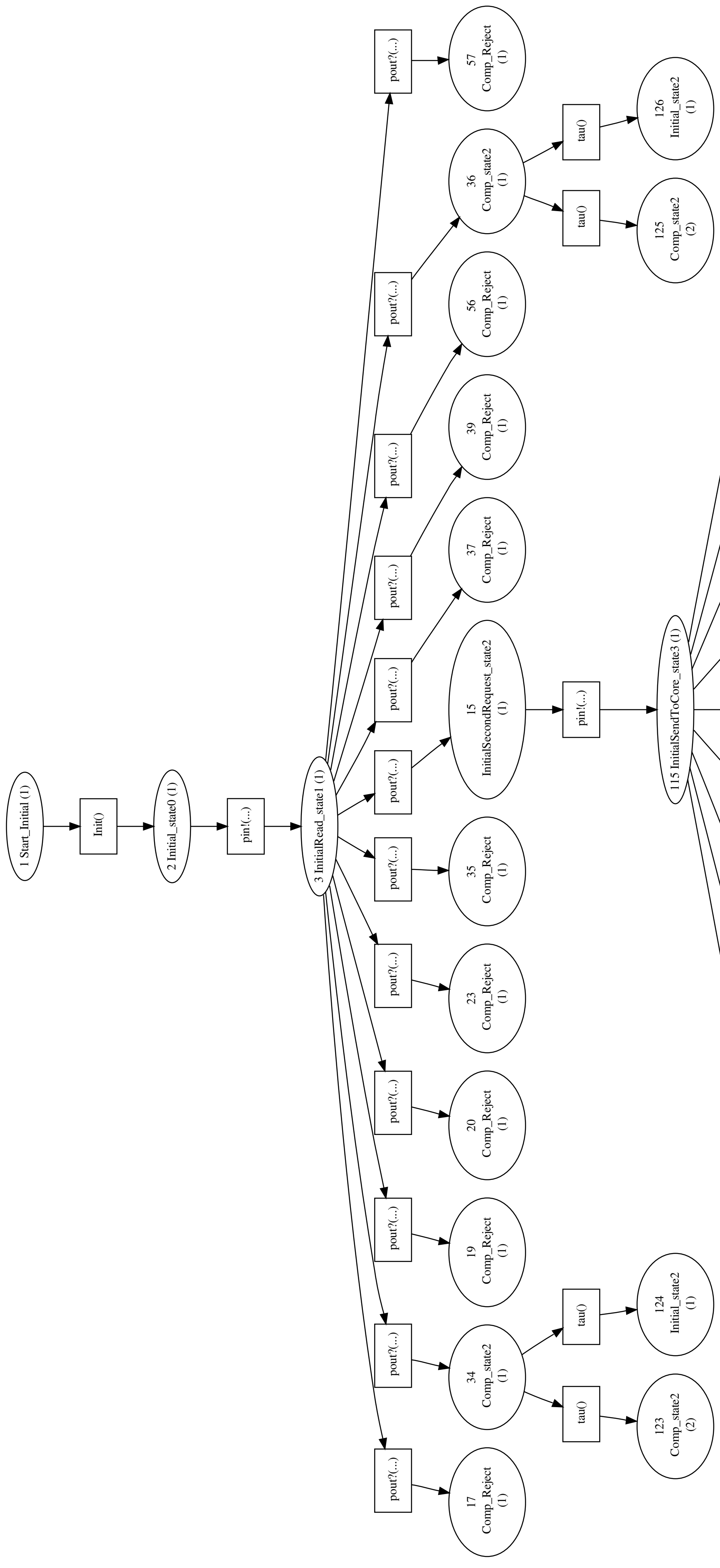


Figure 6.5: This graph shows the structure of one example test case for the SIP Registrar. The used test purpose was "register_notfound.tp".

reached any more. The test case in Figure 6.5 contains 13 of these states. Some of them are labelled by the IOSTS location "Comp_Reject". The "Reject" phrase originates from the used test purpose and indicates behaviours that are not targeted by the test purpose. Note that each *Reject* state is a leaf of the tree. The SES labelled by the number 123 is an *Inconclusive* state as well. The closed product, which has been symbolically executed, contains a self-loop for the IOSTS location "Comp_state2" that is not guarded. Since we used an upper bound of value 2 for the generation of this test case, loop unfolding is stopped after the second time an SES labelled with the same IOSTS location in calculated.

The test case does not contain any *Fail* verdicts. They are identified during test case execution if the IUT sends some input to the test case that is not recognized in the current state.

6.3 Conference Protocol

The Conference Protocol, which has been used previously for the assessment of formal testing tools like TGV [48] or TorX [9], is the third example to evaluate our approach. It is a simple communication protocol for a chat application, whose central part is the so-called Conference Protocol Entity (CPE). The CPE serves as chat client and allows the user to participate in conferences and exchange messages with all other members of the same conference. Each user has a nickname and may only participate in one conference at a time. In the following, the most important aspects of the Conference Protocol will be presented. A more detailed description of the Conference Protocol is provided online[6] by the University of Twente, Netherlands.

Service Primitives and PCOs The Conference Protocol provides four service primitives that can be performed at the Conference Service Access Points (CSAPs). They are called *join*, *leave*, *dataind*, and *datareq*. The communication between the conference partners is accomplished via *dataind* for receiving messages and *datareq* for sending messages. The partners in a conference may change dynamically, since users may *join* conferences and *leave* them again. The Protocol Data Units (PDUs) processed by these four service primitives are transferred via the User Datagram Protocol (UDP). Its service primitives *udp_datareq* for sending data and *udp_dataind* for receiving data are performed at the Underlaying Service Access Point (USAP). CSAP and USAP form the PCOs (Points of Control and Observation) for a tester.

Protocol Behaviour Initially, the user is only allowed to join a conference. Once a user is participant of a conference, he/she may send and receive messages or leave the conference. Each *join* is sent to all potential conference partners in the form of a *join-PDU*, which is acknowledged by all participants of the joined conference by sending an *answer-PDU* back to the sender. Each performed *datareq* causes the CPE to construct a *data-PDU*, which is sent to all conference partners. The CPEs that receive the *data-PDU* perform a *dataind* in order deliver the received message to the user. If a CPE performs a leave, a *leave-PDU* is constructed and sent to all conference partners. Note that all PDUs are sent via UDP's service primitive *udp_datareq* and received by UDP's service primitive *udp_dataind*. Since UDP is a connectionless and unreliable service, it has to be considered that data packages may get lost or may be received out of sequence. The web page describing the Conference Protocol[7] presents an adequate handling of these cases.

6.3.1 Specification

The size of the Conference Protocol Entity (CPE) specification is described by Table 6.5. The IOSTS specification comprises approximately 310 lines of code written in STG format (see the STG Reference

[6]`http://fmt.cs.utwente.nl/ConfCase/` (last visit 2009-09-27)
[7]`http://fmt.cs.utwente.nl/ConfCase/v1.00/description/confprot.html` (last visit 2009-09-27)

Specification (Conference Protocol)		
Category	**Metric**	**Value**
general information	lines of code	~ 310
	# system parameters	0
	# variables	15
	# locations	15
	# transitions	34
actions	# different input actions	2
	# different output actions	2
	# different internal actions	1
# messages of different actions	minimum	2
	maximum	6
	average	5
# operators in guards (logical, arithmetic, and comparison operators)	minimum	0
	maximum	12
	average	6
# logical operators in guards (not, and, or)	minimum	0
	maximum	7
	average	3

Table 6.5: This table describes the size of the Conference Protocol specification.

Manual[8]). It consists of 15 locations connected via 34 transitions. The CPE specification does not have any system parameters, but needs 15 variables for storing its data. It supports 5 different actions: two input actions, two output actions, and one internal action. At an average, each action carries 5 messages. The maximum number of messages for one action is 6. One action has only 2 messages. At an average, the guards of the transitions include 6 operators, whereof 3 are logical operators. The highest number of operators in one guard is 12. The highest number of logical operators in one guard is 7. There are also transitions that are not guarded.

6.3.2 Test Case Generation

Test Purposes

Table 6.6 gives an overview of the size of the (intermediate) results produced throughout test case generation. The different test purposes are described at the top of the table. Three metrics are used to describe their size: (1) the lines of code needed to specify them in STG format, (2) the number of locations, and (3) the number of transitions that each test purposes contains. An average test purpose for the Conference Protocol comprises about 313 lines of STG code. It consists of 13 locations connected via 71 transitions.

Closed Products

The second part of Table 6.6 describes the closed products generated by STG. Five metrics state the size of each closed product. They count the number of (1) locations, (2) transitions, (3) system parameters, (4) variables, and (5) different message names. Note that the values given for the number of transitions do not include duplicates, since redundant transitions are removed from the closed products before they are symbolically executed. At an average, each closed product consists of 59 locations connected via

[8]`http://www.irisa.fr/prive/ployette/stg-doc/stg-web_4.html` (last visit 2009-09-27)

Test Case Generation (Conference Protocol)												
		answer_leave_tp	**answer_pdu_tp**	**double_msgs_tp**	**join_tp**	**join_leave2_tp**	**join_leave_tp**	**join_leave_join_tp**	**receive_msg2_tp**	**send_msg2_tp**	**send_msg_tp**	**Average**
Test Purposes	lines of code	~ 240	~ 180	~ 210	~ 190	~ 320	~ 270	~ 350	~ 440	~ 450	~ 475	~ 313
	# locations	11	9	10	10	14	12	15	16	16	17	13
	# transitions	54	40	47	44	72	61	79	100	101	108	71
Closed Products	# locations	39	31	35	30	56	60	70	79	91	96	59
	# transitions	260	171	232	105	276	306	348	608	655	568	353
	# system parameters	0	0	0	0	0	0	0	0	0	0	0
	# variables	15	15	15	15	15	15	15	15	15	15	15
	# messages	12	12	12	12	13	13	13	13	13	13	13
Symbolic Execution Trees	# SES	4011	1431	7311	4287	9011	14259	> 62500	51043	36355	>75500	26572
	# satisfiable *Accept* states	128	72	328	96	256	512	≥ 0	1536	1216	≥ 548	469
	upper bound	3	3	3	3	3	4	4	3	3	3	3
Generated Test Cases	# SES	18	12	13	14	27	86	–	27	26	–	28
	# SES in longest path	12	9	10	10	15	17	–	15	15	–	13
	# SES in shortest path	3	3	3	3	3	3	–	3	3	–	3
	# Pass	2	1	1	1	1	1	–	2	2	–	1
Test Case Generation Time [mm:ss]	STG	00:47	00:28	00:41	00:29	00:25	00:27	00:32	01:20	01:35	01:57	00:52
	regular termination	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	–
	New Approach	03:26	00:48	11:01	07:02	21:17	58:15	> 36 h	~ 12 h	~ 3.5 h	> 54.5 h	~ 10.75 h
	STG	00:02	00:02	00:03	00:01	00:04	00:03	00:05	00:08	00:12	00:08	00:05
	own implementation	03:24	00:46	10:58	07:01	21:13	58:12	> 36 h	~ 12 h	~ 3.5 h	> 54.5 h	~ 10.75 h
	regular termination	yes	yes	yes	yes	yes	yes	no	yes	yes	no	–

Table 6.6: This table describes the process of generating test cases for the Conference Protocol. It states the size of the intermediate and final results as well as the time needed to generate test cases for different test purposes.

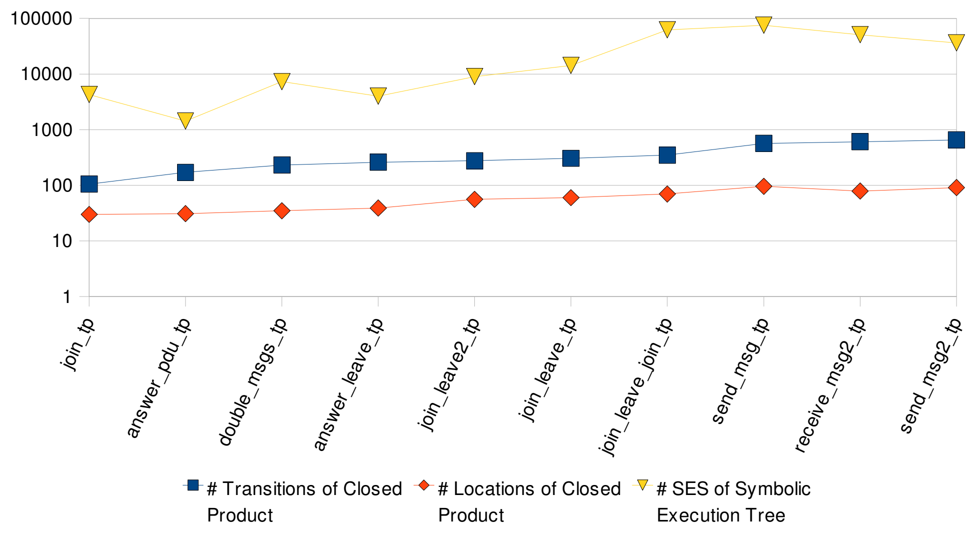


Figure 6.6: For the Conference Protocol example, the size of the closed products in terms of the number of transitions or locations are only weakly correlated with the size of the resulting symbolic execution trees in terms of their number of SES.

353 transitions. None of them has system parameters. Each closed product has 15 variables and contains 12 to 13 different message names.

Symbolic Execution Trees

The symbolic execution trees for the closed products are described in the middle of Table 6.6. Note that not all symbolic execution trees could be completely calculated in reasonable time. The test case generation process for the test purposes “join_leave_join_tp” and “send_msg_tp” was aborted after 36 hours and 54.5 hours respectively. At this program state, the symbolic executions were not finished yet. Hence, the values in Table 6.6 give a lower bound for the size of the symbolic execution trees as well as for the time needed to generate test cases for the affected test purposes. The average values take the upper bounds into account as normal values. At an average, each symbolic execution tree contains 26572 symbolic extended states (SES), whereof 469 are satisfiable *Accept* states. The single values show only a weak correlation between the size of the closed products in terms of the number of transitions or locations and the size of the resulting symbolic execution trees in terms of their number of SES (cf. Figure 6.6).

The upper bound that limits the depth of loop unfolding during symbolic execution (see Section 4.3.2) has been 3 for all test purposes except for “join_leave_tp” and “join_leave_join_tp”. For these two test purposes, the value of 3 was too low. Since the loop unfolding was stopped too early, no satisfiable *Accept* states were calculated and hence it was not possible to generate any test cases. Instead we used an upper bound of value 4, which lead to a successful test case generation for “join_leave_tp”. For the test purpose “join_leave_join_tp”, we do not know if this value was sufficiently high, since the symbolic execution could not be finished within a reasonable amount of time and no satisfiable *Accept* states have been found until the program was stopped. Note that a higher value for the upper bound would cause the program to need even more time. Furthermore, the test case generation for the test purpose “send_msg_tp” was aborted after about 54.5 hours. But in this case we know for sure that an upper bound of value 3 is sufficient, since there were already 548 satisfiable *Accept* states found before the symbolic execution was stopped. For the two test purposes “join_leave_join_tp” and “send_msg_tp”, the values in Table 6.6 describe the size of the symbolic execution trees at the moment of interrupting the program.

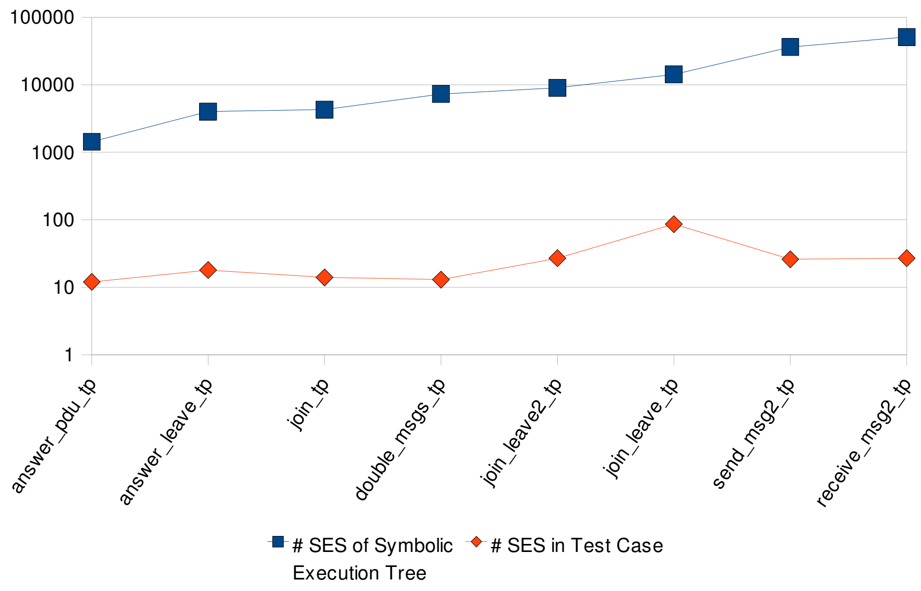


Figure 6.7: The size of the symbolic execution trees and the size of the generated test cases in terms of their number of SES do not correlate for the Conference Protocol example.

Generated Test Cases

Table 6.6 also describes the test cases that were extracted from the symbolic execution trees according to the algorithm defined in Section 4.4.1. Therefor, four metrics are used: (1) the total number of SES in each test case, (2) the number of SES in the longest path, (3) the number of SES in the shortest path, and (4) the number of *Pass* verdicts in each test case. For two test purposes ("join_leave_join_tp" and "send_msg_tp") the symbolic execution could not be finished in reasonable time. Hence, no test cases could be generated for them. At an average, each of the generated test cases consists of 28 SES, whereof one is an *Accept* states, i.e., a *Pass* verdict. The average number of SES in the shortest paths of the symbolic execution trees is 3. The longest paths consist of 13 SES on average. The size of the symbolic execution trees and the size of the generated test cases do not correlate (cf. Figure 6.7).

Test Case Generation Time

Finally, the amount of time that was needed for test case generation by STG and the amount of time that was taken by our approach are listed at the bottom of Table 6.6. The elapsed time for test case generation with our approach is broken down into two parts: (1) the amount of time that is needed by STG to complete the test purpose, to calculate the product, and to close the product, and (2) the amount of time that is consumed by the newly implemented parts of our approach, i.e., by the symbolic execution of the closed product and the test case selection from the resulting symbolic execution tree. Note that all values which are not listed with a specifically unit are given in minutes and seconds. All values were measured from program executions without printing debug output and without exporting intermediate results, i.e., the parsed IOSTS as well as the generated symbolic execution tree, into `dot`/`pdf` format. The rows labelled by *regular termination* state whether STG and the program which implements our approach terminated successfully, i.e., whether they generated test cases.

For the Conference Protocol example, STG did significantly perform better than our approach. First of all, STG was able to generate test cases for all test purposes. Our approach did not manage to generate test cases for two test purposes in reasonable time. Furthermore, STG generated each test case in less than two minutes. By contrast, our approach took up to twelve hours to successfully create a test case. Figure 6.8 illustrates the test case generation times of STG and our approach. Note that all values are given in seconds and that the ordinate is scaled logarithmically.

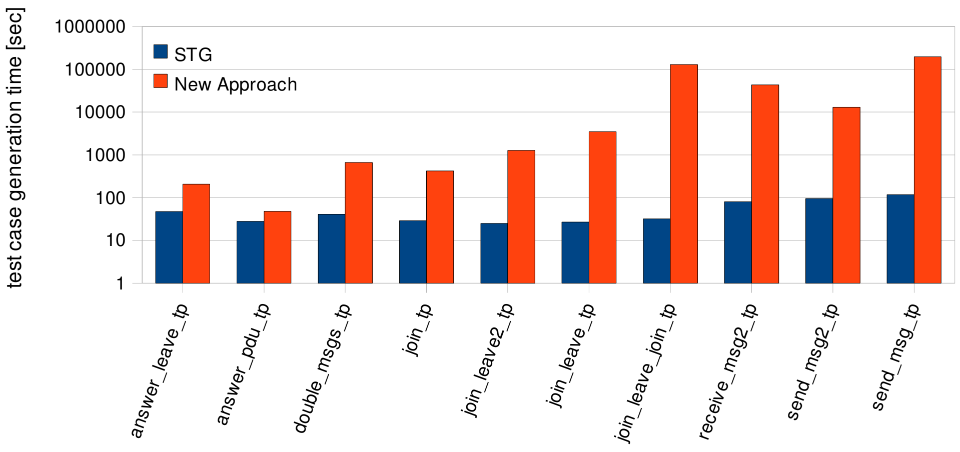


Figure 6.8: This diagram illustrates the time consumed by STG compared to the time needed by our approach to generate test cases for the Conference Protocol.

Example Test Case

Figure 6.9 serves as an example of a generated test case. The symbolic extended states and symbolic transitions of the test case have the same structure as already explained in Section 4.3.3. For this example, the path conditions and symbolic values for each SES as well as the symbolic action messages and message mappings for each symbolic transition have been removed in order to present the test case on a single A4 page. The depicted test case corresponds to the test purpose "answer_pdu_tp". It tests whether the IUT correctly implements the answering of received *join_PDUs*.

Test cases generated with our tool contain implicit verdicts, which have to be made explicit during test case execution. If the test execution ends in an *Accept* state, which is specified through the test purpose, then the verdict is *Pass*. The test case depicted in Figure 6.9 contains one *Pass* verdict represented by the SES labelled with "Joined_Accept". All other leaves in the tree structure represent the verdict *Inconclusive*, since no *Accept* state can be reached any more. The test case in Figure 6.9 contains two of these states. They are labelled with "3 Initial2Idle_state0 (2)" and "21 Joined_state3 (1)". The test case does not contain any *Fail* verdicts. They are identified during test case execution if the IUT sends some input to the test case that is not recognized in the current state.

6.4 Summary and Conclusion

This chapter has presented the results of applying our approach to three examples. The specifications of the SUTs differ heavily in their size. Whereas the Triangle Type Checker specification (see Table 6.1) is very small, the specification of the Conference Protocol Entity (see Table 6.5) is significantly larger, but still not as large as the SIP Registrar specification (see Table 6.3). This applies for almost all of the metrics used for measuring the size of the specifications.

The size of the used test purposes for each example was described. Additionally to the size of the generated test cases, the size of the closed products and the symbolic execution trees was presented for each test purpose. Furthermore, the time needed for test case generation was compared to the time needed by STG to calculate test cases. Table 6.7 summarizes the test case generation metrics for all used examples by stating the minimum, maximum, and arithmetic average values. Note that the minimum and maximum values of the test case generation times state the minimum and maximum values of each single metric. Hence, the sum of the minimum/maximum value of the time needed by the reused parts of STG and the time needed by our implementation does not need to be the same as the minimum/maximum amount of time needed by our approach as a whole.

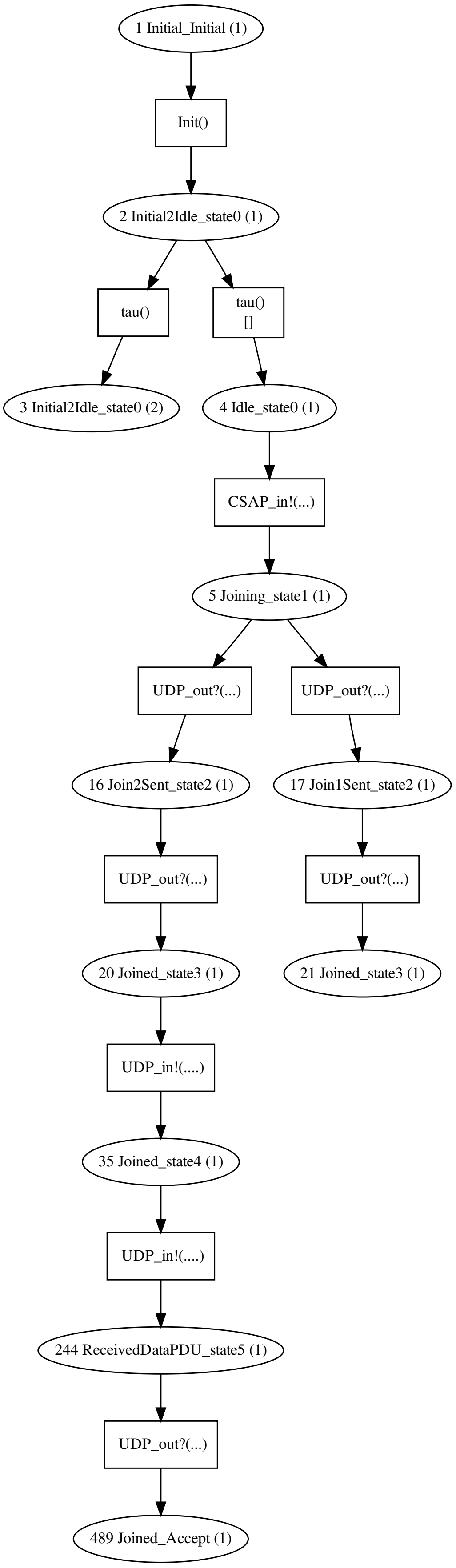


Figure 6.9: This graph shows the structure of one example test case for the Conference Protocol. The used test purpose was “answer_pdu_tp”.

Comparison of the Minimum, Maximum, and Average Values of the Test Case Generation Metrics										
		Triangle Type Checker			**SIP Registrar**			**Conference Protocol**		
		min	max	ϕ	min	max	ϕ	min	max	ϕ
Test Purposes	lines of code	~ 20	~ 20	~ 20	~ 30	~ 90	~ 53	~ 180	~ 475	~ 313
	# locations	3	3	3	4	7	5	9	17	13
	# transitions	2	2	2	5	18	9	40	108	71
Closed Products	# locations	7	7	7	30	113	65	30	96	59
	# transitions	10	10	10	183	562	306	105	655	353
	# system parameters	0	0	0	2	2	2	0	0	0
	# variables	3	3	3	28	28	28	15	15	15
	# messages	3	3	3	11	11	11	12	13	13
Symbolic Execution Trees	# SES	6	9	9	38	566	210	1431	> 75500	26572
	# satisfiable *Accept* states	1	1	1	1	100	28	0	1536	469
	upper bound	2	2	2	2	2	2	3	4	3
Generated Test Cases	# SES	6	9	9	6	38	20	12	86	28
	# SES in longest path	4	5	5	4	7	5	9	17	13
	# SES in shortest path	4	4	4	4	4	4	3	3	3
	# Pass	1	1	1	1	24	8	1	2	1
Test Case Generation Time [sec]	STG	1.68	3.85	2.57	2 min	30 min	14 min	25	117	52
	New Approach	0.91	2.17	1.22	3.93	14.35	7.31	48	> 54.5 h	10.75 h
	STG	0.09	0.23	0.17	1.31	3.37	2.05	1	12	5
	own implementation	0.7	2.08	1.05	2.62	11.19	5.26	46	> 54.5 h	10.75 h

Table 6.7: This table compares the minimum, maximum, and average of the test case generation metrics for the Triangle Type Checker, the SIP Registrar, and the Conference Protocol.

Although the specification for the SIP Registrar was by far the largest of the three used specifications, the average size of the closed products of the Conference Protocol was not much smaller than the average size of the closed products of the SIP Registrar. This can be led back to the used test purposes, which were relatively small for the SIP Registrar (about 53 lines of STG code on average) and rather large for the Conference Protocol (approximately 313 lines of STG code on average).

Notably are the values describing the size of the symbolic execution trees. At an average, a symbolic execution tree for the Conference Protocol has 26572 SES, whereas a symbolic execution tree for the SIP Registrar has only 210 SES on average. Hence, the Conference Protocol causes huge symbolic execution trees compared to the symbolic execution trees of the SIP Registrar, while their closed products were almost of the same size. Although the size of the closed products and the symbolic execution trees show a correlation except for two outliers for the SIP Registrar (see Figure 6.2), the Conference Protocol shows that this correlation cannot be established in general (see Figure 6.6). An explanation for this circumstance may be the internal structure of the closed products. By trend, IOSTS with many loops result in larger symbolic execution trees. Obviously, the size of the symbolic execution trees correlates with the amounts of time needed for test case generation. However, it does not correlate with the size of the generated test cases in general (see Figure 6.7), although a correlation could be identified in the SIP Registrar case study (see Figure 6.3).

Compared to STG, our approach works well for the SIP Registrar for which STG cannot generate any test cases. However, it suffers from performance problems when being applied to the Conference Protocol. While STG uses BDDs, our approach is based on a combination of symbolic execution and SAT solving. In the field of model checking, it has been observed that BDD-based model checkers and SAT-based model checkers are often able to solve different classes of problems. Hence, BDD-based techniques and SAT-based techniques are assumed to be complementary [21; 23]. The two test case generation techniques presented in this work seem to behave similarly, although our approach is not solely based on SAT solving.

STG uses BDDs, which are limited in their applicability due to their memory requirements. In the worst case, BDDs grow exponentially with the size of the Boolean formula they are representing. Hence, the most common failure related to BDDs is “running out of memory” [17]. This conforms to our experience made by applying STG to the SIP Registrar specification, which uses very large guards and a considerable number of variables and messages (see Table 6.3).

By contrast, our approach employs a combination of symbolic execution and SAT solving and does not perform well for the Conference Protocol, which comprises smaller guards and less variables (see Table 6.5). Nevertheless, its symbolic execution trees are huge and the longest paths in the resulting test cases contain significantly more SES than the longest paths in the SIP Registrar test cases (see Table 6.7). Again, an analogy to model checkers exists: It has been observed that BDD-based techniques show a good performance for designs with deep counterexamples, whereas SAT-based model checking techniques are more effective for designs with shallow counterexamples [3].

7 Related Work

7.1 Related Work on Transition Systems

Since the field of model-based testing (MBT) is very large, several surveys were already conducted in order to point out the main trends in MBT. Aichernig et al. [1] as well as Hierons et al. [44] give a good overview about different modelling techniques and corresponding test case generation approaches. The main models presented by at least one of these two surveys are:

- Contract-like specifications/Model-based formal specifications
- Abstract data types/Algebraic languages
- Labelled Transition Systems (LTS)
- Kripke structures and temporal logic for model checking
- Finite State Machines (FSMs)
- Hybrid systems
- Dataflow models

Testing with model checkers is studied in more detail by Fraser et al. [38]. Lee and Yannakakis [55] give a deeper insight on testing with FSMs. Broy et al. [19] are also concerned with MBT. They focus on MBT of reactive systems and concentrate on two system models: FSMs and LTS. Moreover, they deal with model-based test case generation techniques and present up-to-date tools and case studies. In the following, several more specific publications that are closely related to this work will be presented.

As already explained in Chapter 4, this work's approach is based on the tool STG, which is described in detail in Chapter 3. STG reuses various ideas implemented in the tool TGV [48]. Both tools use specifications and test purposes as starting point for test case generation. TGV uses Input Output Labelled Transition Systems (IOLTS) as underlying model. As already mentioned in Section 2.2, the IOSTS model used by STG extends IOLTS with parameters and variables. Hence, TGV does not work on a symbolic level like STG. TGV implements the *ioco* conformance relation, which means that quiescence is taken into account. STG does not consider any quiescence (see Section 2.3). The test generation processes of both tools are very similar, since STG adopts TGV's approach to be applicable on a symbolic level.

Aichernig, Weiglhofer et al. [2] employed the TGV tool for conformance testing of a Session Initiation Protocol (SIP) Registrar, which is also part of our case study (see Section 6.2). The used LOTOS specification was developed by Weiglhofer [78].

Another testing tool that implements the *ioco* conformance relation is TorX [9]. During test case generation, TorX randomly walks through the model, which is represented by an LTS. In case of a nondeterministic choice, several paths are followed in parallel. Just like STG and TGV, TorX supports test purposes, which are a set of traces over visible labels of the LTS. The algorithm implemented by TorX has been modified in order to work with symbolic values by Frantzen et al. [37].

AGATHA [12] is a test case generation tool that accepts specifications written in different languages, e.g., UML, SDL, and STATEMATE. Each specification is transformed into the internally used model IOSTS, which is also employed in this work (see Section 2.2). In the literature about AGATHA, IOSTS are sometimes referred to as EIOLTS (Extended Input Output Labelled Transition Systems). AGATHA also uses symbolic execution as discussed in the following section.

7.2 Related Work on Symbolic Execution

Lúcio and Samer [56] address different techniques employed for test case generation. Besides theorem proving and model checking, symbolic execution is one of them. The application of symbolic execution in the area of testing was already proposed in the 1970s, when King [54] developed the interactive program testing and debugging tool EFFIGY. In the following, more recent work in the field of testing with symbolic execution will be presented.

The AGATHA tool set [12], which was already mentioned in Section 7.1, is applying the same technique as this work: symbolic execution of IOSTS. In contrast to our approach, AGATHA does not use the concept of test purposes. While we symbolically execute a part of the system specification selected via test purposes, AGATHA automatically generates test cases from a whole system specification, which is symbolically executed. Since AGATHA is intended for the validation of system specifications, it attempts to achieve exhaustive symbolic path coverage. In this way, system properties shall be verified by proving them for all paths of the system.

Gaston et al. [40] present an extension of the AGATHA tool set, which aims at helping the tester to identify test purposes. The approach defines test purposes as subtrees of the symbolic execution tree. Test purposes may still be chosen manually by the user, though two coverage criteria are introduced to automatically derive test purposes. According to the selected test purposes, test cases are generated.

The AGATHA tool as well as our work employ symbolic execution of a model of the SUT (system under test) to generate tests and hence deal with black-box testing. By contrast, all of the publications presented in the following are engaged in white-box testing, i.e., they work directly on the system's code.

Symstra [79] is a test generation framework that is focused on object-oriented systems. It uses symbolic execution in order to generate unit tests that reach a high branch and intra-method path coverage.

Khurshid et al. [52] present a framework for correctness checking and test input generation based on symbolic execution and model checking. A source to source translation for program instrumentation is defined in order to allow standard model checkers to symbolically execute the program. In general, model checkers are used to check whether or not a certain property is violated by a system. If a violation is found, a counterexample is computed. The basic idea of testing with model checkers is to use counterexamples as test cases. The survey about testing with model checkers conducted by Fraser et al. [38] gives a deeper insight to the subject.

The testing approaches of this work and of the tools that were mentioned so far use pure symbolic execution. Due to potential non-determinism, a huge number of possible execution paths may be followed, which leads to vast symbolic execution trees. There exist many approaches for testing that employ a combination of concrete and symbolic execution to reduce the number of execution paths. This approach is often referred to as *dynamic symbolic execution* or *concolic execution* (*conc*rete symb*olic* execution). Boyer et al. [16] were probably the first to apply this technique. Their interactive tool SELECT computes symbolical constraints on particular program runs selected by the user.

DART (Directed Automated Random Testing) [41] is a fully automated tool for software testing. It extracts the program's interface and generates a test driver for random testing. During random testing, DART computes symbolic constraints on the current program path, which are used to generate further test inputs forcing the program to follow another path. The goal is to test all possible program paths. DART is limited, since it only supports linear integer constraints. It has been extended in the tools CUTE (Concolic Unit Testing Engine) and jCUTE [69], which support any data type including pointers. An approach similar to the one of DART and CUTE/jCUTE is implemented in the tool EXE [22]. DART, CUTE, and EXE were developed to test C programs. jCUTE is a version of CUTE to test Java programs.

Another tool that employs dynamic symbolic execution is Pex [73]. It generates test input data for *parameterized unit tests* with the goal to achieve a preferably high code coverage. Griesmayer et al. [42] extend dynamic symbolic execution to be applicable to distributed and concurrent systems specified in the modelling language *Creol*.

The algorithm DASH [6] does not directly address the generation of test cases. Its goal is to prove programs similarly to SLAM [4]. Test generation operations and symbolic execution are used to decide where to abstract and how to refine the introduced abstractions. DASH's symbolic execution algorithm relies on techniques inspired by CUTE [69].

As can be seen from the above presented publications, symbolic execution is an active area of research. Although the symbolic execution of programs has already been introduced in the 1970s [53], it has been revived and applied to executable models. Moreover, many recent testing tools employ both concrete and symbolic execution.

8 Concluding Remarks

8.1 Summary

In this work, we addressed automated conformance testing and presented two symbolic test case generation approaches. Both are based on IOSTS (Input Output Symbolic Transition Systems), which extend Input Output Labelled Transition Systems (IOLTS) by the use of variables and parameters. System specifications as well as test purposes, which are used in conformance testing to specify what aspects of the system have to be tested, are modelled as IOSTS. In this way, test cases can be generated without enumerating the specification's state space. The resulting test cases are symbolic and can be made executable by instantiation of their variables.

The first approach was already implemented in the tool STG (Symbolic Test Generator). The single steps of its test case generation process have been described in detail. Basically, STG completes the specified test purpose and calculates the product between the specification and the completed test purpose. Subsequently, the product is closed, i.e., transitions labelled by internal actions are eliminated. Theoretically, the next step of the test case generation approach is to determinize the closed product. However, STG does not implement this operation. The IOSTS which has been generated so far is already a valid test case. Nevertheless, it is bigger than necessary and may include unreachable locations. Thus, a test case selection procedure is performed. STG's approach uses reachability and coreachability analyses for this task. Finally, the selected test case has to be made input-complete.

However, this approach shows weaknesses for some kinds of systems. In our case, STG was not able to generate test cases for the SIP Registrar. Thus, an alternative way of generating test cases from IOSTS has been developed and presented in this work. Since only a part of the existing STG tool set causes problems, the new approach does not start from scratch, but was designed to reuse basic functionality concerning IOSTS from STG. Above all, the new approach replaces STG's problematic parts concerning the test case selection, which is based on reachability analyses. Instead, it employs symbolic execution of IOSTS, which requires SAT solving.

The new test case generation process presented in this work consists of the following steps: It starts like STG's test case generation procedure and calculates the product between the specification and the completed test purpose. Then, the closure operation is applied to the calculated product. Subsequently, our new test case selection is performed. It symbolically executes the closed product generated by STG and selects a test case from the resulting symbolic execution tree.

This work also covered the prototype implementation of our new approach, which has been used to show the applicability of our test case generation procedure in the course of three case studies. The first case study covered a very small example called Triangle Type Checker, which was used for illustration purposes throughout this work. The other two case studies dealt with protocols: the Session Initiation Protocol (SIP) and the Conference Protocol. In particular, the former is of industrial relevance since it is a popular signalling protocol used in various applications, e.g., VoIP. The conclusions that can be drawn from the case studies will be discussed in the following section.

8.2 Conclusion

One of the main contributions of this work is the implementation of our test case generation approach based on the symbolic execution of IOSTS and SMT solving. The developed tool provides the same interface as STG, which means that specifications written for STG can be reused without any modification. A further advantage of our implementation is its modular architecture. For example, the integration of another SMT solver than Yices should cause no big effort.

By the development of our tool, we were able to contribute to the field of symbolic conformance testing by evaluating both test case generation approaches presented in this work. In the course of our three case studies, it has turned out that none of the approaches is better than the other. Each of them has its strengths and weaknesses. Our approach works well for the SIP Registrar. This means that we have achieved our key objective to generate symbolic test cases for the SIP Registrar, for which STG cannot generate any test cases. However, we have also shown the limits of our approach. It suffers from performance problems when being applied to the Conference Protocol, for which STG achieves good results. This circumstance indicates that the application areas of the two approaches are different.

In this regard, we could identify analogies to the field of model checking, where it has been observed that BDD-based model checkers and SAT-based model checkers are often able to solve different classes of problems [21; 23]. STG uses BDDs, while our approach employs a combination of symbolic execution and SAT solving. Although our approach is not solely based on SAT solving, the experiences from model checking saying that BDD-based and SAT-based techniques are somehow complementary seems to apply for the investigated test case generation techniques as well.

Furthermore, it has been noticed that BDD-based model checkers show a good performance for designs with deep counterexamples, whereas SAT-based model checkers are more effective for designs with shallow counterexamples [3]. In the field of test case generation, we have observed that STG, which uses BDDs, shows a better performance for examples that result in test cases with deep paths. By contrast, our approach, which is based on a combination of symbolic execution and SAT solving, achieves better results for examples that result in test cases of a smaller depth.

Finally, we come to the conclusion that there is no silver bullet for the automated generation of test cases. Each approach is somehow limited. Fortunately, they seem to complement each other so that a solution can be found for each problem, either by applying the one or the other approach.

8.3 Future Work

Although we have achieved good results for the SIP Registrar (see Section 6.2), the application of our approach to the Conference Protocol Entity specification (see Section 6.3) has shown that there is still room for improvement. Some proposals about future work will be given in the following.

At present, the algorithm for test case selection (see Section 4.4.1) is not very sophisticated. It generates just one test case by arbitrarily selecting one *Accept* state of the symbolic execution tree. One improvement could be to implement TGV's policy [48] and to support both, (1) the generation of just *one* test case per test purpose and (2) the generation of *all* possible test cases for one test purpose.

Furthermore, if the user chooses to generate just one test case although the symbolic execution tree contains more than one *Accept* state, then the decision about which test case will be generated should be up to the user. He/she could decide from which *Accept* state the test case selection starts. In this way, the user would be able to influence test case selection and help to generate more useful test cases.

In the case of generating just one test case, an opportunity to improve the performance of our algorithm would be to use an on-the-fly approach. The full symbolic execution of the IOSTS could be stopped when the first satisfiable *Accept* state has been calculated. Henceforward, only transitions which are part of the generated test case according to the rules presented in Section 4.4.1 have to be executed. In this way, it would be possible to minimize the effort for symbolic execution.

Another point of future work is the integration of guard strengthening. STG strengthens the guards during its coreachability phase in order to avoid test cases from following paths that lead to an *Inconclusive* state (see Section 3.1.5). Although an inconclusive verdict cannot be prevented entirely, STG has better chances of reaching the goal of the test purpose than we have.

Currently, we cannot guarantee to generate test cases that do not depend on internal choices of the tester. Hence, some determinization heuristic like it was planned but not implemented by STG (see Section 3.1.4) would be desireable.

At the moment, our approach is implemented in the form of a prototype, which could be enhanced in many ways. Besides an improved memory usage, the invocation of STG could be integrated into our Java application. In this way, the shell script currently used to combine STG and our Java implementation would be unnecessary. Thus, the usability of our tool could be facilitated. The user would only have to bother with one Java application, which could provide a graphical user interface (GUI).

Moreover, the test case generation tool of this work shall be combined with an already existing software for translating UML State Charts into IOSTS [72]. By allowing the user to define system specifications and test purposes via UML State Charts, a better support for the modelling of large systems is provided. Furthermore, a tool for executing the generated test cases is under development while writing this thesis. Hence, a framework for covering the model-based testing process including model creation, test case generation, and test case execution could be provided.

The work done so far in the course of this thesis already contributes to the efficiency and acceptance of model-based testing. Nevertheless, as can be seen from the above proposals, it is far from being completed and stimulates further research in the field of automated test case generation.

Bibliography

[1] Bernhard Aichernig, Willibald Krenn, Henrik Eriksson, and Jonny Vinter. State of the Art Survey - Part a: Model-based Test Case Generation. Technical report, Institute for Software Technology (IST), Graz University of Technology, June 2008. (Cited on pages 1, 6 and 102.)

[2] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods. In *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM'07)*, pages 215–226. IEEE Computer Society, 2007. (Cited on pages 13, 87 and 102.)

[3] Nina Amla, Robert P. Kurshan, Kenneth L. McMillan, and Ricardo Medel. Experimental Analysis of Different Techniques for Bounded Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003. (Cited on pages 101 and 106.)

[4] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. *SIGOPS Operating Systems Review*, 40(4):73–85, 2006. (Cited on page 104.)

[5] R. M. Balzer. EXDAMS: EXtendable Debugging and Monitoring System. In *Proceedings of the Spring Joint Computer Conference (AFIPS '69)*, pages 567–580, 1969. (Cited on page 59.)

[6] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from Tests. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 3–13. ACM, 2008. (Cited on page 104.)

[7] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2nd edition, 1990. (Cited on pages 4 and 5.)

[8] Ferenc Belina and Dieter Hogrefe. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, 1989. (Cited on page 11.)

[9] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal Test Automation: A Simple Experiment. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems: Method and Applications*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer Academic Publishers, 1999. (Cited on pages 6, 93 and 102.)

[10] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for Test Case Generation. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer, 2004. (Cited on pages 42 and 43.)

[11] Gilles Bernot. Testing Against Formal Specifications: A Theoretical View. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT '91)*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 1991. (Cited on page 12.)

[12] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic Test Generation with AGATHA. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 591–596. Springer, 2003. (Cited on pages 102 and 103.)

[13] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman, 1999. (Cited on page 2.)

[14] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987. (Cited on page 11.)

[15] Cari Borrás. Overexposure of Radiation Therapy Patients in Panama: Problem Recognition and Follow-Up Measures. *Rev Panam Salud Publica*, 20(2/3):173–187, 2006. (Cited on page 1.)

[16] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245. ACM, 1975. (Cited on pages 59 and 103.)

[17] Daniel Brand. Verification of Large Synthesized Designs. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pages 534–537. IEEE Computer Society Press, 1993. (Cited on pages 54 and 101.)

[18] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975. (Cited on page 1.)

[19] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*. Number 3472 in Lecture Notes in Computer Science. Springer, 2005. (Cited on page 102.)

[20] Stanislaw Budkowski and Piotr Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987. (Cited on page 11.)

[21] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Improving SAT-Based Bounded Model Checking by Means of BDD-Based Approximate Traversals. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE 2003)*, page 10898. IEEE Computer Society, 2003. (Cited on pages 101 and 106.)

[22] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10:1–10:38, 2008. (Cited on page 103.)

[23] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating BDD-Based and SAT-Based Symbolic Model Checking. In *Frontiers of Combining Systems (FroCoS 2002)*, volume 2309 of *Lecture Notes in Computer Science*, pages 49–56. Springer, 2002. (Cited on pages 101 and 106.)

[24] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Automated Test and Oracle Generation for Smart-Card Applications. In *Smart Card Programming and Security: Proceedings of the International Conference on Research in Smart Cards (E-smart 2001)*, volume 2140 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001. (Cited on pages 42 and 43.)

[25] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Tool for Generating Symbolic Test Programs and Oracles from Operational Specifications. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 301–302, 2001. (Cited on pages 6 and 23.)

[26] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A Symbolic Test Generation Tool. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer, 2002. (Cited on page 23.)

[27] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976. (Cited on page 59.)

[28] Lori A. Clarke and Debra J. Richardson. Applications of Symbolic Evaluation. *Journal of Systems and Software*, 5(1):15–35, 1985. (Cited on page 60.)

[29] P. David Coward. Symbolic execution systems – a review. *Software Engineering Journal*, 3(6):229–239, 1988. (Cited on page 59.)

[30] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997. (Cited on page 5.)

[31] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975. (Cited on page 35.)

[32] Mark Dowson. The ARIANE 5 Software Failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997. (Cited on page 1.)

[33] Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver. Available at `http://yices.csl.sri.com/tool-paper.pdf` (last visit 2009-09-27). (Cited on pages 64, 74 and 75.)

[34] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. In *Proceedings of the 9th International Symposium on Graph Drawing (GD 2001)*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer, 2001. (Cited on page 49.)

[35] Alain Faivre and Christophe Gaston. Test generation methodology based on symbolic execution for the Common Criteria higher levels. In *2nd MoDeVa workshop - Model design and Validation*, 2005. (Cited on page 64.)

[36] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A Symbolic Framework for Model-Based Testing. In *First Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES 2006 and RV 2006)*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006. (Cited on pages 1, 2 and 6.)

[37] Lars Frantzen, Jan Tretmans, and Tim A.C. Willemse. Test Generation Based on Symbolic Specifications. In *Formal Approaches to Software Testing (FATES 2004)*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005. (Cited on page 102.)

[38] Gordon Fraser, Paul Ammann, and Franz Wotawa. Testing with Model Checkers: A Survey. Technical Report SNA-TR-2007-P2-04, Institute for Software Technology, TU-Graz, Austria, 2007. (Cited on pages 102 and 103.)

[39] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997. (Cited on page 35.)

[40] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic Execution Techniques for Test Purpose Definition. In *18th International Conference on Testing of Communicating Systems (TestCom'06)*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006. (Cited on pages 61, 63 and 103.)

[41] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223. ACM, 2005. (Cited on page 103.)

[42] Andreas Griesmayer, Bernhard K. Aichernig, Einar Broch Johnsen, and Rudolf Schlatte. Dynamic Symbolic Execution for Testing Distributed Objects. In *Second International Conference on Tests and Proofs (TAP'09)*, volume 5668 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2009. (Cited on page 103.)

[43] Penny Grub and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing, 2nd edition, 2003. (Cited on page 1.)

[44] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2):9:1–9:76, 2009. (Cited on pages 5 and 102.)

[45] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. (Cited on page 41.)

[46] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. (Cited on page 64.)

[47] International Organization for Standardization. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework*. International Standard ISO/IEC 9646. International Organization for Standardization, Geneva, Switzerland, 1991. (Cited on pages 8, 9, 11 and 12.)

[48] Claude Jard and Thierry Jéron. TGV: Theory, Principles and Algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005. (Cited on pages 6, 8, 67, 68, 93, 102 and 106.)

[49] Bertrand Jeannet, Thierry Jéron, and Vlad Rusu. Model-Based Test Selection for Infinite-State Reactive Systems. In *5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*, volume 4709 of *Lecture Notes in Computer Science*, pages 47–69. Springer, 2006. (Cited on pages 13, 14, 23, 42 and 48.)

[50] Bertrand Jeannet, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Symbolic Test Selection Based on Approximate Analysis. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2005. (Cited on page 13.)

[51] Joost-Pieter Katoen. Labelled Transition Systems. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 615–616. Springer, 2004. (Cited on pages 6, 13, 16 and 22.)

[52] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003. (Cited on page 103.)

[53] James C. King. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233. ACM, 1975. (Cited on pages 59, 60 and 104.)

[54] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976. (Cited on pages 59 and 103.)

[55] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996. (Cited on page 102.)

[56] Levi Lucio and Marko Samer. Technology of Test-Case Generation. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 323–354. Springer, 2004. (Cited on page 103.)

[57] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995. (Cited on page 5.)

[58] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979. (Cited on pages 1, 14 and 83.)

[59] Claude Petitpierre. Synchronous Active Objects Introduce CSP's Primitives in Java. In *Proceedings of the Conference on Communicating Process Architectures (CPA 2002)*, volume 60 of *Concurrent Systems Engineering Series*, pages 109–122. IOS Press, 2002. (Cited on pages 39 and 41.)

[60] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2008. (Cited on page 6.)

[61] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer, 1986. (Cited on page 11.)

[62] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Wiley, 2000. (Cited on page 72.)

[63] Alexander Pretschner and Martin Leucker. Model-Based Testing - A Glossary. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 607–609. Springer, 2004. (Cited on page 8.)

[64] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004. (Cited on pages 2, 3 and 5.)

[65] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. (Cited on page 75.)

[66] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC3261: SIP: Session Initiation Protocol. RFC Editor, 2002. Available at `http://www.rfc-editor.org/rfc/rfc3261.txt` (last visit 2009-09-27). (Cited on page 87.)

[67] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An Approach to Symbolic Test Generation. In *Proceedings of the 2nd International Conference on Integrated Formal Methods (IFM 2000)*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2000. (Cited on pages 6, 13, 14, 16, 17, 18, 19, 20, 29, 31 and 33.)

[68] Vlad Rusu, Hervé Marchand, Valéry Tschaen, Thierry Jéron, and Bertrand Jeannet. From Safety Verification to Safety Testing. In *Testing of Communicating Systems (Proceedings TestCom'04)*, volume 2978 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2004. (Cited on page 13.)

[69] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Cited on pages 103 and 104.)

[70] J. M. Spivey. *The Z Notation: a Reference Manual*. International Series in Computer Science. Prentice-Hall, 1989. (Cited on page 11.)

[71] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. (Cited on page 2.)

[72] Christopher Thurnher. Model Transformation from UML State Machines to Input/Output Symbolic Transition Systems. Master's thesis, Institute of Information Systems - Vienna University of Technology, 2008. (Cited on pages 1 and 107.)

[73] Nikolai Tillmann and Jonathan de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. (Cited on page 103.)

[74] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, 1992. (Cited on pages 8, 9 and 11.)

[75] Jan Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996. (Cited on page 22.)

[76] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. (Cited on pages 5, 13 and 22.)

[77] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, 2006. (Cited on pages 1, 2, 4 and 5.)

[78] Martin Weiglhofer. A LOTOS formalization of SIP. Technical Report SNA-TR-2006-1P1, Competence Network Softnet, Graz, Austria, December 2006. (Cited on pages 87 and 102.)

[79] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer, 2005. (Cited on page 103.)

[80] Eléna Zinovieva-Leroux. *Symbolic Test Generation for Reactive Systems with Data*. PhD thesis, University of Rennes 1, Rennes, France, 2004. (Cited on pages 5, 8, 19, 20, 21, 22, 24, 26, 27, 28, 33, 34, 35 and 37.)