Simplifying Network Testing: Techniques and Approaches Towards Automating and Simplifying the Testing Process

by

Constantinos Djouvas

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of Doctor of Philosophy,
The City University of New York
2009

THE CITY UNIVERSITY OF NEW YORK
DEPARTMENT OF COMPUTER SCIENCE

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

Prof. Nancy D. Griffeth

Date — Chair of Examining Committee

Prof. Theodore Brown

Date — Chair of Examining Committee

Prof. M. Ümit Uyar

Prof. Ping Ji

Prof. Nancy Lynch

Supervisory Committee

Abstract

Simplifying Network Testing: Techniques and Approaches Towards Automating and Simplifying the Testing Process

by

Constantinos Djouvas

The dramatic increase of companies and consumers that heavily depend on networks mandates the creation of reliable network devices. Such reliability can be achieved by testing both the conformance of individual protocols of an implementation to their corresponding specifications and the interaction between different protocols. With the increase of computer power and the advances in network testing research, one would expect that efficient approaches for testing network implementations would be available. However, such approaches are not available due to reasons like the complexity of network protocols, the need for different protocols to interoperate, the limited information on implementation because of proprietary codes, and the potentially unbounded size of the network to be tested.

To address these issues, a novel technique is proposed that improves the quality of the test while reducing the time and effort network testing requires. The proposed approach achieves these goals, by automating the process of creating models to be used for validating an implementation. More precisely, it utilizes observations acquired by monitoring the behavior of the implementation for the automatic generation of models. In this way, generated models can accurately

represent the actual implementation. Thus, testing is reduced to the problem of verifying that certain properties hold on the generated model. This work presents algorithms that efficiently create models from observations and shows their effectiveness through the presentation of three different examples.

In addition, the difficulty of validating models using theorem provers is addressed. To address this issue, techniques available in the literature are utilized and approaches that assist testers with completing proofs are proposed. Results suggest that the complexity of making proofs using theorem proving can be reduced when models are members of the same class, i.e., their structure can be predicted.

A final problem this work addresses is that of scale, i.e., the impracticality or even impossibility of testing every possible network configuration. To address this problem, the concept of "*self-similarity*" is introduced. A self-similar network has the property that can be sufficiently represented by a smaller network. Thus, proving the correctness of a smaller network is sufficient for proving the correctness of any self-similar network that can be represented by this smaller one.

Acknowledgments

First and foremost I would like to thank my adviser Prof. Nancy D. Griffeth for her guidance and help throughout my doctoral studies. Her high standards and expectations as well as her determination helped me stay focused and gave me the opportunity to work on and solve extremely interesting problems. I am also grateful for all the personal support she offered me all these years.

I thank Prof. M. Ümit Uyar who taught me the fundamental ideas and approaches for network testing. I deeply appreciate all his help, ideas, comments and above all his out-of-the-box way of thinking.

I would also like to thank Myla Archer and Elizabeth Leonard for their help on theorem proving. Special thanks to Myla for all her help during my visits at the Naval Research Lab.

I thank Prof. Nancy Lynch for her help and great ideas on addressing scalability issues in network testing and supporting the idea of self-similar networks. In addition, I would like to express my gratitude for her detailed comments on this document.

I gratefully acknowledge the members of my dissertation defense committee: Prof. Nancy D. Griffeth, Prof. M. Ümit Uyar, Prof. Ping Ji, and Prof. Nancy Lynch.

Finally I would line to thank my colleagues Jiang Wu and Yuri Cantor for collaborating with me in all the stages of my research.

This material is based upon work supported by the National Science Foundation under Grant No. 0435130. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Στους γονείς μου, Στα αδέλφια μου και Στη Σταυρούλα μου.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

Rapid technological improvements lead to the integration of computer use in practically every aspect of daily life. Computers are not isolated machines, only able to perform a particular range of operations in a small amount of time; rather, they are combined into network systems and thus become gateways to the whole world. As a result, reliable network devices are of great importance. Such reliability can be achieved by testing network devices for their correct implementation and interoperability.

The importance of *network testing*, i.e., the examination of whether network devices operate properly, has been recognized since the first implementations of network systems. As early as 1978, Jan Hajek [28] proposed a technique for testing the conformance of an implementation to its specification and Colin West [58] suggested a technique that tested the correctness of a specification. Ever since, network testing has been an active research field, wherein many different

approaches have been proposed.

Current best practices in network testing require the development of a formal model that describes the functionality of the Implementation Under Test (IUT). This model is created by hand and must accurately represent the expected behavior of the implementation, as described by its specification. A model that describes the functionality of the IUT is used for generating a minimal set of test cases. The generated test cases are then utilized to determine the implementation's conformance to its specification [41]. Different techniques have been proposed and used effectively for the systematic generation of test cases from models [1, 12, 27]. However, their use uncovered some difficulties which include the following:

- The creation of a formal model by hand is difficult and time-consuming. For instance, in Bishop et al.'s case [12], it took nine man-years to create a rigorous model of the TCP specification.

- Formal models are rarely available to testers prior to testing. This is mainly because of the time required for building such models.

- The model itself may be incorrect unless carefully validated.

- An accurate model, i.e., a model that accurately represents the implementation, is not necessarily a complete model, so entire classes of errors could be missed.

- Usually the model of a specification has a huge state space, sometimes making the problem intractable.

- There is no answer to the question: How likely is it for an implementation to work in an environment different from the one under which it was tested? It is impossible to test in a lab every possible configuration and topology and verify that the implementation will interoperate properly with every other implementation that a real network can have.

These questions and problems provide the motivation and rationale for this dissertation. In what follows, I refer specifically to the purpose and objectives of my work.

1.2 Purpose and Research Objectives

This work aims at the development of a novel network testing technique that solves some of the problems encountered in current network testing approaches. First and foremost, solutions to problems related to the manual creation of a formal model used for testing an implementation must be proposed. Thus, the ultimate goal of this work is the development of a technique that automatically generates and validates models of different implementations. This can be achieved by automatically creating a model that accurately represents a network implementation through the utilization of observations acquired by monitoring its behavior. When such model is available, testing a network implementation is reduced to the problem of validating the generated model, i.e., proving that some properties defined in its specification hold on the generated model. Taking into consideration that a completely automated solution is close to impossible, the goal of this work is the development of an approach, where much of the process of constructing a model by observing a live network can be automated, hence reducing the overall effort

of creating models.

The goal of automating the process of creating and validating a model of a network implementation, raises further questions. The first focuses on approaches the tester can use for efficiently and effectively extracting useful information from observations acquired by monitoring the network. Information extracted by this process will be considered as words accepted by the generated model. In addition, since it is close to impossible to observe all possible behaviors of an implementation, approaches that make accurate predictions about possible behaviors of the implementation must be adopted. Furthermore, approaches for building models by utilizing the extracted information must be investigated. Finally, approaches for validating the generated model, namely *Model Checking* or *Theorem Proving*, must be utilized. Model checking is automatic and thus, this work only focuses on theorem proving, and more precisely, on how the process of proving that properties hold on models can be simplified.

A different problem that network testing faces is scalability, given that it is not always possible to test every possible configuration. This work aims at proposing techniques that can be used for increasing testers' confidence that an implementation will work properly in environments and configurations different from those in which it was tested.

1.3 Contribution

This work contributes to the field of network testing by proposing and developing a novel network testing technique that improves the quality of network testing while reducing the test time. For achieving this goal, techniques that can be

used for building representative models of network implementations through the utilization of observations[1] are proposed. The proposed approach is related to the exponential time problem of machine identification defined by Moore in [49]. In order to overcome the prohibitive complexity of machine identification, heuristics are proposed, which can be used for making predictions about behaviors of the implementation. As a result, more complete models (i.e., models that contain more behaviors) can be created using fewer observations.

For creating interesting models, one must extract interesting subsequences of the original sequence of messages observed. This suggests that the user should be able to slice the trace in any arbitrary way. Such arbitrary slicing is achieved by defining an extended form of relational algebra query, the *Parameterized Queries*. *Parameterized Queries* allow parameters in the place of attributes or domain values. Techniques proposed for efficiently calculating their results can be used for data mining. Actually, this work extends the relational aggregation operation *data cube*, introduced by Gray et al. in [24].

In addition, techniques that automatically identify and add loops to the generated model are proposed. By adding loops to a model, the number of its states is reduced, while its completeness increases. This is because loops allow behaviors that are already in the model to be repeated many times. Techniques proposed here are general, and one can use them for identifying potential loops is a set of words or in finite state machines.

Techniques that simplify the process of validating models using theorem provers are also investigated. Such simplification is achieved by identifying a class of mod-

[1]For the context of this work, observations are defined as sequences of messages exchanged between different devices (known as *traces*), acquired by monitoring the network.

els amenable to more efficient proofs. Also, experiments contacted and presented in this work suggest that more efficient proofs can be achieved when the structure of a model is predictable. Thus, examples provided in this thesis can be utilized by researchers for identifying different classes of models amenable to efficient proofs. In addition, they can be used as a guideline for proposing specific techniques for simplifying theorem proving on models of specific classes.

Another problem in network testing that is addressed is that of scale. In this thesis, an approach that overcomes this obstacle in specific networks is presented, which is nevertheless independent of the approach used for testing each of these networks. More precisely, this work presents a way of finding a representative network to test, instead of testing all network members of a class of networks. In this way, by testing the representative network is equivalent to testing all members of the class of networks to which it belongs. The proposed approach can be used for identifying the smallest possible representative network.

Finally, network administrators might find this work of interest because it presents a way for analyzing and understanding the behavior of networks by utilizing the tools implemented for building models through observations. More precisely, as part of this work, Automatic Generation of Automata for TEsting (AGATE) was developed, an open-source software tool that automatically creates models of network implementations. Models are graphically represented as state machines. Based on the experience gained from its use in real networks, models generated by AGATE are extremely useful in understanding how network components work, given that it isolates specific interactions among specific components. Analyzing behaviors of specific components using graphical representation of state machines is much easier than trying to do the same by using the entire trace.

1.4 Thesis Structure

The rest of this thesis is organized as follows:

Chapter 2 starts with a thorough review of work related to both software and network testing. First, I present definitions and work proposed for software testing. Those were the first approaches to testing and thus some software testing techniques are utilized in network testing. Then, I introduce some basic concepts for testing network implementations, and I provide definitions related to state machines and their utilization in network testing. Techniques proposed for network testing that rely on state machines are presented in section 2.4. In addition, in chapter 2, I present some background information necessary for this work. More precisely, I present IOA/TIOA, the formal language used for modeling systems. Finally, in section 5.2, I present the two different approaches, Model Checking and Theorem Proving, one can use for establishing that some properties hold on models.

In chapter 3, I describe a novel network testing approach which is a precursor of this thesis. The goal of this approach is to simplify network testing by automating the testing process. More precisely, a representative model of the Implementation Under Test (IUT) is automatically created through observations acquired by observing its behavior. Assuming that the generated model accurately represents the IUT, by proving that properties hold on the generated model, one also proves that the same properties hold on the IUT. The main objective of this thesis is the automatic creation of a model that represents the IUT through the utilization of network observations. In addition, in this chapter I present a comprehensive example that illustrates all the steps of the algorithm that creates models from

observations. The reader can use this example as an introduction to chapters 4 and 5, where I describe in detail each step of the algorithm that creates formal models from observations.

Chapter 4 describes the different steps required for creating a model represented as a state machine from a set of observations. The first step analyzes and breaks each observed sequence of messages into smaller subsequences. Each subsequence captures a permissible behavior of a component (or a collection of components) of the implementation. After that, I present an approach that abstracts each subsequence created. This approach facilitates the creation of more complete models by predicting possible behaviors of the IUT through the utilization of the available observations. Each abstracted subsequence is treated as a word that the generated state machine will accept. I then present the algorithm used for building a state machine that accepts this set of words. Finally, I present an approach that automatically identifies and adds loops to the generated state machine.

For validating the generated model, and thus proving the correctness of the implementation, the generated model should be translated into a formal language. In chapter 5, I first present the translator that produces an automaton expressed in a formal language, Timed Input Output Automata (TIOA) in our case, from a state machine. Then, I explain how the generated TIOA model can be validated against some properties using theorem proving. In addition, I present some techniques that can be used for simplifying theorem proving when models belong to a specific class, i.e., models that were created through the use of the process described above.

As mentioned earlier, the first step for creating a model from a set of observa-

tions is to create "interesting" subsequences of the original sequence of messages observed. In chapter 6, I present *Parameterized Query*, an approach that allows the creation of arbitrary subsequences of the original sequence. In addition, in the same chapter, I present a technique that can efficiently calculate the results of a parameterized query.

Chapter 7 presents Automatic Generation of Automata for TEsting (AGATE), an open-source software tool that automatically creates TIOA models from network observations. It describes every functionality implemented in AGATE at the time of this thesis's completion and explains how these functionalities can be used for automatically building models from network observations.

In chapter 8, network testing scalability, a general problem of network testing, is addressed. This problem is independent of the network testing approach used and stems from the fact that it is impossible to test every possible scenario and configuration in a lab. To partially overcome this obstacle, an approach is proposed for finding a representative network to test, instead of testing all networks that belong to the same class. Thus, by testing the representative network one actually tests all the members of the class to which it belongs.

Conclusions and suggested future work are presented in chapter 9.

Chapter 2

Related Work and Background

In this chapter, I cover work related to testing by reviewing various techniques that have been proposed for both software and network testing. First, in section 2.1, I present some software testing techniques, considering that current network testing approaches utilize techniques proposed for software testing. After introducing some basic concepts for testing network protocols in section 2.2, I provide some useful definitions related to state machine approaches in section 2.3. Techniques based on state machines are presented in section 2.4, considering that a large number of current network testing techniques utilize those.

In addition, I use this chapter to present background information necessary for this work. More precisely, in section 2.5, I present IOA/TIOA, the formal language I use for modeling systems. Finally, in section 2.6 (Formal Verification), I present the two different approaches one can use for establishing the correctness of a model, that is, Model Checking and Theorem Proving.

2.1 Software Testing Techniques

"Program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra 1972: 6) [18]. Based on this statement, one can define testing as the procedure of uncovering bugs that a system has. This definition illustrates testing's difficulty level. Since one cannot prove that an implementation is correct and considering that the number of bugs is unknown, the answer to the question "Is this system acceptable?" admits only statistical answers.

Testing of any kind of software has been an active research area for many years. Determining whether a system has no bugs demands exhaustive testing, i.e., all possible input must be tested. This raises one of the greatest difficulties that testing faces, that is, issues related to time. Testing every possible input takes for ever, unless there are only finitely many inputs. Even for very small and trivial programs, the number of different inputs can be huge [50, 59], thus making exhaustive testing intractable.

Because exhaustive testing is unrealistic, research has focused on techniques that maximize the number of errors found by a finite number of test cases, i.e., create "good" test cases [50]. These techniques can be divided into two categories, *White Box Testing* and *Black Box Testing.* In white box testing, the examination of the internal structure of the program is possible [50], that is, the source code is available in the testing procedure. However, such techniques are not applicable in network testing, where in most of the cases the code is proprietary. This is the reason that white box testing techniques are out of the scope of this review.

The latter category, black box testing, treats the program as a black box and it is thus more relevant to network testing. The goal is to uncover the behavior

of the "black box". Such techniques concentrate on identifying circumstances in which the program does not behave according to its specifications [50].

The two most important categories of such techniques are:

1. *Equivalence partitioning*: the input cases are split into subsets of "similar" inputs. The system either fails for all or for no member of each set.

2. *Boundary value analysis*: it identifies test cases that explore boundary conditions (e.g., if one knows that the program should behave differently when the value of a variable i is i <1, i=1 and i >1, then values from all these three categories must be tested).

Another important issue to be solved in software as well as network testing is stopping conditions; namely, when the testing procedure should stop. Myers in [50] provides the two most common criteria used by testers as stopping conditions:

1. When the testing time expires.

2. When all the test cases give no errors.

According to Myers, both criteria are useless. The first criterion, by far the most common one, is useless in terms of measuring the quality of a test, since it is satisfied without doing anything. The second one can be satisfied by using bad test cases, that is, cases that are unable to uncover any error. However, the second criterion can become more useful, if the test cases are generated through equivalence partitioning or boundary values.

Another criterion that can be used as a stopping condition is to predefine the number of errors that must be found before testing is terminated. However, this

criterion requires an accurate estimation of the number of errors that a program can have, which makes it hard, if not impossible, to be accurately used.

Littlewood and Wright in [45, 46] answered the question of "what new testing requirement should be imposed upon the system following a failure of a well defined test?". In other words, what should be the stopping conditions of a test after previous tests have failed. For answering this question, they defined p to be the probability of a failure upon demand, where demand is a set of circumstances that require the system to take an action. They framed these requirements in the Bayesian context as a pair (p_0,a), such that $P(p < p_0) \geq 1 - a$. Providing p_0 and a, they present formulas that can be used for calculating the required number of demands, given the number of previous failures that must be applied on the system for achieving the desirable confidence bound.

The above discussion suggests that finding a statistically correct stopping condition for testing is close to impossible. It also suggests that "correct" stopping conditions mostly depend on the experience and the intuition of the tester.

2.2 Testing Network Protocols: Validation, Verification, and Interoperability

Any activity in a network that involves at least two communicating remote entities is governed by a network protocol [38]. A protocol defines the rules that every participant in a network activity should follow. A proposed protocol becomes a standard, after a period of development and several iterations of reviews by experts [14]. The two most well known organizations that issue such standards for internet are the Institute of Electrical and Electronics Engineers (IEEE) [34] and

the Internet Engineering Task Force (IETF) [55]. Both follow similar procedures based on the method stated above, i.e., through many iterations of reviews and improvements. IEEE has a dedicated division following this process for creating standards, known as IEEE-SA [32]. The process followed by IETF is open to the public and is known as the RFC process [56], where everyone can participate in proposing improvements on a protocol.

After the community is convinced that a proposed protocol is correct, a standard for that protocol is issued, containing its specification. Based on the specification, any vendor can create its own implementation of the particular protocol.

Errors may exist in both parts of the procedure, i.e., in the specification and/or the implementation phase. First, the specification can be wrong, which means that even a correct implementation of that specification would also be incorrect; on the other hand, the implementation might not be a correct translation of the specification. Even in the unlikely event that a wrong translation of a wrong specification causes a correct implementation, one should consider it as wrong implementation. In all the above cases, the final product will differ from what was intended.

Since errors may exist in the specification and the implementation, testing should be performed in both. Based on the definition that West suggests in [58], the term *validation* is used to describe the testing procedure that determines whether or not the specification is sound and its logical structure complete. *Verification* is concerned with the comparison of particular aspects of the implementation's behavior specified in the specification. In this review, I only address verification techniques, because this work aims in developing a network testing technique for verifying implementations and not for validating standards.

Even in the case of correct implementation, a network device might still not operate properly. This is because there is another unpredictable and difficult to uncover source of "misbehaving" network devices, which can be attributed in the inability of independently developed devices to interact with one another properly (for example "big endian" and "little endian" machines). *Interoperability*, i.e., the ability of multiple hosts from multiple vendors to communicate efficiently, reliably, and meaningfully [16], must be another concern for contemporary network testing techniques.

2.3 State Machine Based Approaches

Most network testing approaches are based on state machines, i.e., formal methods. In this subsection, I draw on Lee's and Yannakakis' work [41] to present techniques that can be used for testing state machines. I focus on approaches that can be used in network testing and provide only a brief description of approaches not directly applicable to that.

Formal methods are mathematically based techniques for describing systems [60]. They can be used for many purposes, some of which are described by Hall [29] and Bowen et al. [13]. One of the most important functions of formal methods is to verify systems systematically. The formal language used in this work is the IOA/TIOA [22, 36, 47]. Particular features and detailed description of the IOA/TIOA language is provided in a following subsection.

Traditional state machine based approaches are based on deterministic finite state machines.

Definition 2.3.1. A finite state machine (FSM) M is a quintuple M= (*I, O, S,*

δ, λ) where :

- I: is a set of input symbols.
- O: is a set of output symbols.
- S: is a set of states.
- δ: $S \times I \times S$ is the state transition function.
- λ: $S \times I \times O$ is the output function.

When the machine is in state $s \in S$ and receives an input $\alpha \in I$, it moves to the next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$.

State machine testing can be divided into two categories. In the first category, the FSM is completely defined. The question to be answered is: in which state is the FSM? Three different problems appear in this category:

1. *Homing Sequence*: Testing must determine the final state of the FSM after the test.
2. *State Identification*: Testing must identify the initial state of the FSM. An input sequence that solves this problem (if it exists) is called a *distinguishing* sequence.
3. *State Verification*: Testing must verify whether the FSM is at a particular initial state at which it should be. A test sequence that solves this problem (if it exists) is called a *Unique Input Output* (UIO) sequence.

These approaches are not directly applicable to network testing, because the FSM of the actual implementation of a protocol is not available. However, there

are some useful applications that rely on state machines. For example, UIO sequence was used by Aho et al. in [1] for the efficient computation of a test sequence for protocol conformance testing.

The second category of state machine testing is based on black box, i.e., the FSM of a system is unknown. *Conformance Testing* and *Machine Identification* are the two problems that appear in this category.

The first problem, *conformance testing*, utilizes a second FSM that can be derived from the specification of the system. For an implemented system S to be correct, its behavior – as a sequence of input and output messages – should be equivalent to a FSM M of the specification. Using the state transition and the output function of M and observing the Input/Output behavior of S, conformance testing attempts to determine if S is a correct implementation of M [41]. This problem can be solved, if there exists a unique input sequence that distinguishes M from all other FSMs that have the same number of states as M, i.e., a sequence that M produces a different output than that produced by any other FSM having the same number of states. This sequence is called *checking sequence* [41]. An implementation FSM and a specification FSM are equivalent, if and only if they produce the same output having as input the checking sequence. Almost all network testing techniques are based on this method. Such techniques are elaborated in a following subsection.

The second problem is called *Machine Identification* and was proposed by Moore in [49]. This approach attempts to generate a FSM of the implementation based on the I/O behavior of the system. For solving this problem, Moore in [49] makes some strong assumptions. He assumes that the machine to be identified can be represented as a FSM M; that M is strongly connected, that M is reduced, that

M does not change during the experiment; and, the machine's input and output alphabet as well as an upper bound of its number of states are known. Clearly, some of these assumptions are not realistic in network testing. In addition, Moore has proved that the *machine identification* problem is exponential. However, *machine identification* was used by Sabnani et. al. in [40], where the authors proposed an algorithm that locates the differences between the specification's FSM and the implementation's FSM. This shows that despite the complexity of *machine identification*, it can have some useful applications in testing. *Machine identification* is a precursor of the network testing approach proposed in this work.

2.4 Network Testing Techniques

One might expect that network testing can adopt some black box software testing techniques, considering that in both network and software testing the goal is to create test cases to be used for testing a system. However, neither equivalence partitioning nor boundary values, which are black box techniques for software testing, can be used for efficiently and effectively creating test cases in network testing. This is because those were designed for conventional programs where inputs are well defined. However, in networks, inputs are sequences of messages that can arrive in any order.

In addition to difficulties associated with software testing, difficulties related to the nature of networks, such as the one mentioned above (non-determinism) and the problem of interoperability of multiple systems, render state machine based techniques necessary for network testing. Even within these techniques, there exist particular difficulties, which I review in this subsection. I first review the

state explosion problem and describe proposed solutions. Beyond that, I refer to problems that derive from the need to manually create a model of a specification to be used for test case generation.

A difficulty that testing based on state machines faces is that of test case generation, mainly because of the *state explosion* problem. The state explosion problem is described as follows: a model of the specification consisting of all states and all transitions can be extremely large, sometimes large enough to make the test case generation impossible. Since generating every possible test case is impossible, techniques that aim at maximizing the gain of a test, i.e., maximizing fault coverage with minimum cost, have been proposed.

Fecko et al. [20, 21] focused on test case generation in presence of timers. This focus derives from the fact that networks are real time systems with many different timers embedded in each device. In cases where there are conflicts between timers, test case generation becomes unrealizable. This problem is known as the *conflicting timers* problem [20, 21]. The proposed solution relies on the use of simple linear expressions that involve time-related variables and aims at capturing timing dependencies. This shortens the test sequences without compromising the fault coverage [20], since cases that can never occur in the real network because of conflicting timers are never included in a test case sequence.

Griffeth et al. [27] tackle the test generation problem by testing only inputs that trigger interoperation between two devices. This kind of test, which is based on state machines, is known as *interoperability testing*. Interoperability testing differs from conformance testing. The former, tests the interoperations of two systems, whereas the latter tests the conformance of the implementation to its specification. The approach was efficiently and effectively used in relatively small

initial tests; though, more extensive subsequent work uncovered problems including difficulties with formal models (i.e., creating a model was time consuming and not available on time), with model validation, and, finally, with test case generation (i.e., it required management of very large graphs).

Pythia [42] is a tool implemented at Lucent, also aiming at optimizing test case generation. The algorithm that Pythia implements is described in [41]. Given a model, Pythia generates test cases based on three different covering criteria. The first one requires the coverage of all states and transitions. The second is based on group covering, as states are placed into groups and at least one member of each group must be covered (this is also known as the *traveling salesman problem*). The last one requires a subset of variables; at least one occurrence of the transition of every possible combination of values of a subset should be covered.

Pythia was applied to real systems with very good results. As mentioned in [41], it was used for test case generation for the Personal HandyPhone System (PHS), 5eSS Intelligent Network Application Protocol (INAP), and ATM PNNI. Results showed that the first 20% test cases covered more than 70% of all paths.

Bishop et al. [12] approached network testing in a different manner. Instead of focusing on test case generation, they proposed an automatic method for conformance testing of TCP implementations. The particular method suggests the automatic evaluation of traces collected by monitoring the network by utilizing a model of TCP [33] specification. However, this approach requires an accurate model of the specification. Although the results were promising, it took nine man-years to build a rigorous model of the TCP specification. This indicates the difficulties that one can face creating models by hand.

A completely different approach for network testing is used for testing indus-

trial networks, because of the different objectives in that area. As stated in [53], the objectives of testing such networks include:

1. Application response time
2. Application Feature / Functionality
3. Regression
4. Throughput
5. Acceptance
6. Configuration Sizing
7. Reliability
8. Product Evaluation
9. Capacity Planning
10. Bottleneck Identification

These objectives presuppose no actual testing of whether the implementation conforms to the specification. Instead, these approaches check only some practical, performance related issues.

The methodology suggested by Buchanan in [53] for testing those networks identifies six areas. The testing procedure should start with planning. During planning, the objective of testing should be defined and documented. Then, models to be used for creating network load during testing should be decided. The next step is to create the test configuration. During the test, the tester should collect data based on the objectives of the test. For example, if performance is an

objective of the test, response time and data throughput should be collected. Data collected should be interpreted. This part of the test is performed by experts, able to interpret the results. For example, they are able to tell if a particular response time is acceptable or not. The last part of this procedure is data presentation, where all the steps used and results found are summarized and presented.

2.5 Formal Language - IOA/TIOA

The TIOA language [22] is a formal language, which is based on the input/output automaton model [47, 48] designed for describing the behavior of a distributed system.

An I/O Automaton A consists of the following five components:

1. *sig(A)*, a signature, consisting of three disjoint sets of actions: the *input actions in(sig(A))*, *output actions out(sig(A))*, and *internal actions int(sig(A))*. Output and internal actions are *locally controlled*; input actions are controlled by an automaton's environment. The set of all actions in the signature is denoted *acts(sig(A))*.

2. *states(A)*, a nonempty, possibly infinite set of states.

3. *start(A)*, a nonempty subset of *states(A)*, called the *start states*.

4. *trans(A)*, a state-transition relation, contained in *states(A)* $\times$ *acts(sig(A))* $\times$ *states(A)*. It is required that for each state s and input action π, there is a transition (s, π, s').

5. *tasks(A)*, a *task partition*, which is an equivalence relation on the locally

controlled actions of A and which has at most countable many equivalence classes.

An *execution* of an I/O automaton is a sequence s_0, π_1, s_1, ..., s_{n-1}, π_n, s_n where s_0 is a start state and (s_{i-1}, π_i, s_i) is a transition for each $i \geq 1$. An execution can be finite or infinite. The set of executions of an I/O automaton A is denoted as *execs(A)*. *traces(A)* is defined as the set of all sequences π_1, π_2, ..., π_n, ... obtained by removing the states from a sequence in *execs(A)*.

The I/O automaton model includes two important notions. The first, *parallel composition*, allows the construction of complex systems by composing smaller parts. To compose automata, one should consider actions with the same signature in different automata to be the same action. When any component performs an action π, it forces all the components having the same action to perform that action. Thus, for composition to work, automata must be *compatible*. A countable collection $\{S_i\}_{i \in I}$ is *compatible* if for all $i, j \in I$, $i \neq j$, all of the following hold:

1. $int(S_i) \cap acts(S_j) = \phi$
2. $out(S_i) \cap acts(S_j) = \phi$
3. No action is contained in infinitely many sets $acts(S_i)$

Definition 2.5.1. Given a compatible collection $\{A_i\}_{i \in I}$ of automata, the composition $A = \Pi_{i \in I} A_i$ is defined by:

- *sig(A)* is defined by:
  - $out(sig(A)) = \cup_{i \in I} out(sig(A_i))$
  - $int(sig(A)) = \cup_{i \in I} in(sig(A_i))$

- $in(sig(A)) = \cup_{i \in I} in(sig(A_i)) - \cup_{i \in I} out(sig(A_i))$

- *states(A)* $= \Pi_{i \in I}$ *states*(A_i).

- *start(A)* $= \Pi_{i \in I}$ *start*(A_i).

- *trans(A)* is the set of triples (s, π_i, s') such that for all $i \in I$, if $\pi \in$*acts*(A_i) then $(s, \pi_i, s') \in$ *trans*(A_i).

- *tasks(A)* $= \cup_{i \in I}(A_i)$.

The operation $ActHide_{\Phi}(A)$, where Φ is a set of external actions in $sig(A)$, can be used for reclassifying external actions of A as internal. Using this operation, one can prevent actions from being used for communication among different automata. This operation can be useful in automata derived from composition, considering that actions of individual automata previously used for communication between composed automata can be hidden. This operation will make the composed automaton appear as a single entity, where the communication between its components is not externally observable.

The second notion of the I/O automaton model, *simulation relation*, can be used as a sufficient condition to prove a formal relation between two I/O automata. To prove that an automaton A implements another automaton B, one needs to show that for any execution of the automaton A, there is a corresponding execution of the automaton B.

Definition 2.5.2. A simulation relation from an IOA A to an IOA B is a relation $f$on *states(A)* x *states(B)* satisfying the following:

1. If $s \in$ *start(A)*, then $f(s) \cap$ start(B) $\neq \phi$ (start condition).

2. If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and (s, π, s') $\in$ *trans(A)*, then there is an execution fragment α of B starting in state u and ending in some state $u' \in f(s')$ such that $trace(\alpha) = trace(\pi)$ (step condition).

The I/O automaton model also allows the definition of invariant assertions. An *invariant* property of an automaton is any property that should be true in all reachable states of the automaton.

The TIOA (Timed Input/Output Automaton) [36] is an extension of IOA enabling the description and analysis of timed systems. The notion of a *trajectory* is defined, which is used to model the evolution of a collection of variables over an interval of time.

In the network testing technique proposed in this thesis and implemented as part of the AGATE toolkit, the TIOA language is used to model network implementations. Currently, AGATE, which is a tool that automatically creates state machines from network observations, only creates deterministic TIOA models. However, this can be extended to non-deterministic models, considering that this is supported by TIOA. In addition, models created by AGATE do not include time, which should be added in the future.

Tempo [57] is toolkit implemented by VeroModo Inc., and it can be used for either simulating a TIOA model or for translating the TIOA model to either UPPAAL [39] for model checking or TAME [9, 35] for theorem proving. Model checking and theorem proving are described in the following subsection.

2.6 Formal Verification

Formal verification is a process that can be used for justifying that some properties hold on a formal model. Two different categories of formal verification exist in the literature, model checking and theorem proving.

2.6.1 Model Checking

In model checking, the verification of a property is achieved through exhaustive search, i.e., a model checker checks all possible sequences of events and tests at each step that the property is not violated. The use of model checking against the use of theorem proving has two major advantages. The first is that it is automatic. By providing a model and a property, a model checker automatically validates the property on the model. The second advantage of the use of model checking is that it provides counter examples. If a model checker fails, it will provide the execution that violated the property. However, model checking suffers from the *state explosion* problem, since it utilizes brute force approaches.

The most widely used model checker is SPIN [30]. It is a tool designed and developed at Bell Labs during the 1980s and 1990s and since then it has been extensively used with great success. SPIN can be used for both simulating and verifying a model. For model checking though, only the verification mode is useful. The SPIN verifier creates a state machine consisting of all reachable states of the model described in PROMELA and it traverses all the paths of this state machine trying to find an execution that violates a property [30].

UPPAAL [11, 10, 17] is a new model checker that can be used for verification. Its main goal is to alleviate the state explosion problem with a new technique that

reduces the verification problem to that of solving linear constraint systems. In addition, UPPAAL supports timed models, a huge advantage against older model checkers that do not support those.

2.6.2 Theorem Proving

Theorem provers utilize induction for proving that properties hold on models. The advantage of theorem proving is that it does not suffer from the state explosion problem. This is because inductive proofs are independent of the number of states that a model has. However, the use of induction has a disadvantage. Proofs are not fully automatic, since theorem provers require assistance for proving the base case and the induction step of the induction.

The theorem prover used in this work is PVS (Prototype Verification System). It is a system for writing specifications and constructing proofs [51]. The PVS system includes a specification language, a parser, a type checker, and an interactive proof checker. The specification language is based on higher order logic and consists of one or more theories that have to be proved. The type checker is responsible for finding any semantic error that the specification might have. Such errors are reported to the user.

The prover is an interactive environment. At the beginning of the proof the prover creates a set of goals that have to be proved. This is known as the *proof tree*, where the root of the tree represents the property that the user tries to prove, and each branch of the tree represents a subgoal that has to be proved in order to prove the main property. The first branch of the tree represents the base case. After proving the base case, one can assume that the property holds after n steps (induction hypothesis). Assuming that the induction hypothesis holds, the goal

is to prove that no action violates the property, and thus prove that the property also holds at step $n+1$. This can be achieved by proving all the other branches of the tree. Each branch other than the first one represents an action (transition) of the model. By proving these branches, one proves that no action can violate the property during the transition from step n to step $n+1$. The depth of a branch represents the complexity of proving a subgoal, since deeper branches mean that more steps are required for proving them. Interactively, the user tries to break each goal to a collection of simpler subgoals that can be discharged automatically by the primitive proof steps of the prover (i.e., goals that are trivially true). In addition to primitive proof steps, PVS supports more complex proof steps called strategies. PVS comes with some built-in strategies, but the user can define his/her own strategies using primitive proof steps, applicative Lisp code, and other strategies.

TAME (Timed Automata Modeling Environment) [8, 6, 7] is an interface to PVS for proving properties of Input/Output Automata. It provides a template for defining automata that simplifies the encoding of automata specification. In addition, TAME is equipped with a set of standard theories that can be used when making proofs. Finally, and most importantly, special strategies [5] specifically designed for automaton models are implemented for TAME.

The use of TAME strategies [5] has two main advantages. First, they simplify proofs considering that they are specifically designed for automaton models. Second, they mimic human proof steps, which enable the presentation of proofs in human understandable form. This property of TAME strategies allows a tester to both understand every step of the proof and recognize the reasons that cause a proof to fail.

Chapter 3

Proposed Network Testing Methodology

A novel approach for testing network implementations that avoids the error-prone, time-consuming and hard process of building models by hand was proposed by Griffeth and Djouvas in [25, 26]. This approach proposed the automatic generation of models from observed network behaviors. Assuming that the generated model accurately represents the implementation, properties of the implementation can be proved or disproved against this model. Thus, it simplifies network testing by reducing it to the problem of model validation. Figure 3.1 graphically represents the proposed approach.

As in any testing approach, testing starts with a network implementation, the Implementation Under Test (IUT), and a set of requirements that the IUT must meet. Given those, testing becomes an iterative procedure that stops either when a bug is uncovered or when the tester has a high level of confidence that the generated model adequately represents the implementation. Thus, the tester can prove

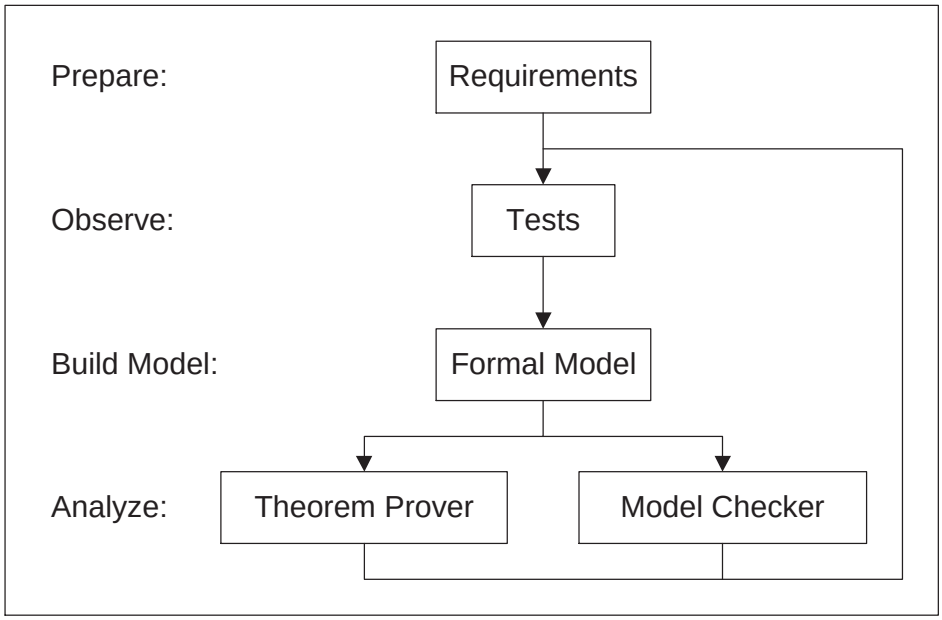

Figure 3.1: The Network Testing Approach.

or disprove its correctness or state conditions under which the implementation will function properly.

The steps that should be followed at all iterations are as follows:

1. *Generate test cases:* At the first iteration the test cases can be created randomly. For all the subsequent iterations, the test cases should be generated based on the model that has been built so far. This is because at any new iteration, it is desirable to capture some new functionality, i.e., a functionality that the IUT can exhibit but the current model of the IUT does not. The tester monitors the network while running tests, collecting all the necessary data for incrementally building the model.

2. *Build model:* Once step 1 has been completed, the tester expands or modifies the model so that it will also reflect the new functionality captured by the new test results.

3. *Analyze:* Both theorem provers and model checkers can be utilized to test the model. The results of these tests can guide the development of predictions necessary for creating new tests. As mentioned earlier, if a property is violated or a theorem cannot be proved, testing can stop.

The main objective of this dissertation is the development of techniques that allow the creation of an accurate model which is as complete as possible. This can be achieved by making accurate predictions about behaviors that have not actually been observed.

Definition 3.0.1. A Model M of an Implementation I generated from a network trace (observation) O is **accurate** if and only if *traces(O)* $\subseteq$ *traces(M)* $\subseteq$ *traces(I)*.

The above definition suggests that a model is accurate if it is a subset of the actual implementation.

Definition 3.0.2. A Model M of an Implementation I generated from a network trace (observation) O is **complete** if and only if *traces(O)* $\subseteq$ *traces(I)* $\subseteq$ *traces(M)*.

Definition 3.0.2 suggests that a model is complete if it is a superset of the actual implementation.

If the current set of observations does not contain enough information in order to create an acceptable model, the tester can run more tests, collect new observations and augment the model. In fact, one might find it convenient to generate and then compose different automata reflecting the behavior for a single implementation. Each automaton maps a different component of the implementation. For example, a DHCP server incorporates two important components: One is

handling the communication with its clients and the other determines which IP address to offer to a client. It is easier to generate two automata, one for managing the communication between client and server and a second for managing IP address allocation. These automata are simple to generate and their composition emulates the behavior of a single implementation.

In general, models that are as close as possible to the IUT are preferable. However, depending on the property of the IUT the tester wants to prove, there are cases for which a subset of all the possible behaviors of the IUT should be preferred. If the property is related to a particular component that can be isolated from every other component of the IUT, then a model that represents only that component is sufficient for proving the property. In this case, the smallest possible subset of the behaviors of the IUT should be preferred, because it will reduce the complexity of both generating and testing the model.

In chapter 4, I present the theory and algorithms used for building a model using network observations. Having a model of the IUT represented as a state machine, the next step is to validate the model against some properties. The first step for doing that is to translate the state machine to a formal language, TIOA in our case, and then use a theorem prover (PVS was used for this work) for formally proving that specific properties hold on the model (See chapter 5). However, before proceeding to chapters 4 and 5, where I formally and in detail explain each step, I use the following section to present a comprehensive example that describes each step of the algorithm that builds models using network observations. This section can be used as an introduction to chapters 4 and 5.

3.1 A Comprehensive Example

In this section, I present a comprehensive example that illustrates each step of the process of creating state machines from observations. However, I start this section through a brief description of DHCP, the protocol I use in this example.

DHCP is a protocol that assigns IP addresses to clients in an IP network. A client that needs an IP address initiates a process for getting one from a DHCP server through a DISCOVER message (multiple DISCOVER messages can be sent by a client if it does not receive a response to the initial DISCOVER message). When a DHCP server receives the DISCOVER message and has at least one available IP address, it responds back with an OFFER message. The server uses this message to offer an IP address to the client. After receiving the offered IP address, the client sends a REQUEST message back to the server. Finally, the server sends an ACK message back to the client for acknowledging that the client can start using the IP address.

In addition to the offered IP address, the ACK message also carries the period of time that the IP address is valid, known as *lease time*. When the lease time expires the server can reassign the IP address to a different client. However, the client has the option to renew the lease time by sending a RENEW[1] message. When the server receives a RENEW message, it responds with an ACK message. Theoretically, the client can keep renewing the offered IP address for ever. The sequence of messages that a DHCP client exchanges with a DHCP server are presented in figure 3.2.

Next, I present a possible trace (observation) that can be generated through

[1]Technically speaking, RENEW messages are REQUEST message. However, this is a term used for distinguish the two messages and simplifying the explanation of the protocol.

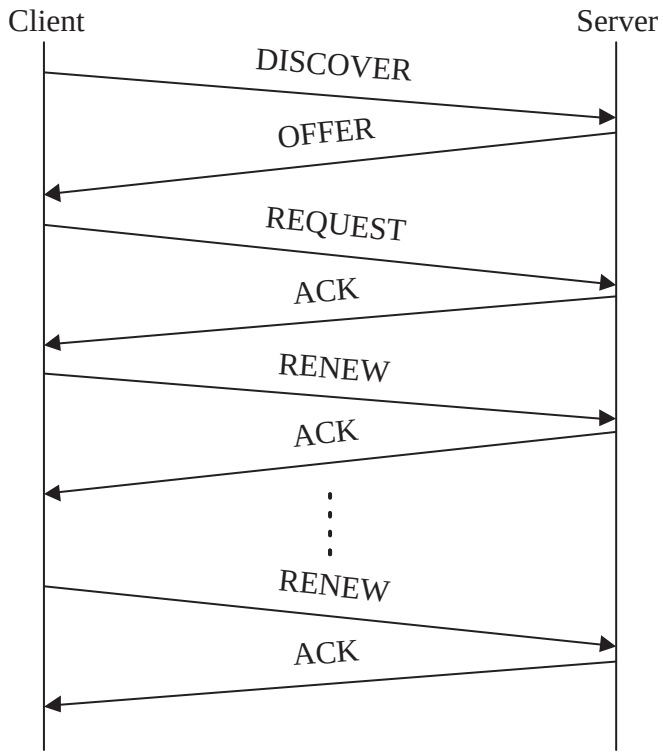

Figure 3.2: Message Sequence Chart of DHCP.

the interaction of a single DHCP server with multiple DHCP clients. For doing that, I use the following notations:

- D: denotes a DISCOVER message.
- O: denotes an OFFER message.
- R: denotes a REQUEST message.
- A: denotes an ACK message.
- Re: denotes a RENEW message.
- Each message is accompanied with two arguments, the *client id* and the *transaction id*. For example *D(i,j)* denotes a DISCOVER message sent by

client i that has a *transaction id j*. Similarly, *O(i,j)* denotes an OFFER message sent to client i that is a response to the discover message sent by client i with *transaction id j*.

Using the notations defined above, assume the trace: D(i,k), D(i,k), D(i,k), O(i,k), D(j,l), R(i,k), D(j,l), A(i,k), O(j,l), R(j,l), A(j,l), Re(i,k), A(i,k), Re(j,l), A(j,l), Re(j,l), A(j,l), D(i,m), O(i,m), R(i,m), A(i,m), Re(j,l), A(j,l), Re(i,m), A(i,m), Re(i,m), A(i,m). This trace will be used in subsequent sections, as an observation utilized for building state machines. In the following section, I present each step of the process of creating a model utilizing the above sequence of messages. The generated model represents the communication component of a DHCP server (handling the communication between a DHCP server and a DHCP client).

3.1.1 Extracting Words from Observations

The first step for creating a state machine from observations is to create subsequences of the original sequence of messages observed. The created subsequences will be treated as words that the generated state machine will accept. In this example, messages are grouped together based on the values of the client id and transaction id fields. More precisely, two messages belong to the same group, if both the client id and the transaction id are the same. Based on this, the following three groups will be created:

- Group 1={ D(i,k), D(i,k), D(i,k), O(i,k), R(i,k), A(i,k), Re(i,k), A(i,k)}.
- Group 2={D(j,l), D(j,l), O(j,l), R(j,l), A(j,l), Re(j,l), A(j,l), Re(j,l), A(j,l), Re(j,l), A(j,l)}.
- Group 3={D(i,m), O(i,m), R(i,m), A(i,m), Re(i,m), A(i,m), Re(i,m), A(i,m)}.

3.1.2 Fields Semantics

Next, semantics must be assigned to fields. This process is driven by the assumption that not all fields have the same importance when building a Session State Machine. Regular fields are fields whose actual value is important for state transition. In this example, such a field is the message type, i.e., DISCOVER, OFFER, etc. In addition, one can assume that the values of some fields have no internal meaning. The client id and transaction id are such fields. This is because the server should not make any distinction between different clients and transactions based only on the client id or transaction id. These fields are declared as Symbolic and only the permutation of values they carry is preserved. More precisely, it is assumed that any permutation of values of these fields in a legal word results in another legal word. For capturing the permutation of values of a particular field, the actual values carried by that field are mapped to symbolic values (they appear as numbers in the following groups). More precisely, the first distinct value carried by a symbolic field is mapped to symbolic value 1, the second distinct value to symbolic value 2, and so forth.

Assigning the above semantics to the three words created in the first step, the following three abstracted words will be created:

- Group 1={ D(1,1), D(1,1), D(1,1), O(1,1), R(1,1), A(1,1), Re(1,1), A(1,1)}.
- Group 2={D(1,1), D(1,1), O(1,1), R(1,1), A(1,1), Re(1,1), A(1,1), Re(1,1), A(1,1), Re(1,1), A(1,1)}.
- Group 3={D(1,1), O(1,1), R(1,1), A(1,1), Re(1,1), A(1,1), Re(1,1), A(1,1)}.

3.1.3 Session State Machine

Using the three abstracted words created in the previous step, the Session State Machine, *SSM*, represented in figure 3.3 will be created.

3.1.4 Adding Loops to the Model

Having a model of the implementation represented as a state machine, the next step is to minimize it. This can be achieved by both adding loops to the state machine[2] and by performing classical minimization techniques [2].

Given the state machine presented in figure 3.3 and the definition 4.4.3, the only loop that exists in the state machine is D(1,1), i.e., a client can send one or more DISCOVER messages at the beginning of a session. By adding this loop to the state machine, the session state machine SSM' presented in figure 3.4 will be created.

The loops algorithm should now be reapplied on session state machine SSM'. By this application, the algorithm recognizes a loop that contains messages <Re(1,1), A(1,1)>. The addition of this loop to SSM' will create session state machine SSM'' presented in figure 3.4.

Further applications of the loop algorithm on SSM'' will discover no more loops. As a result, the algorithm in this particular example stops after three iterations. Also SSM'' is minimized, and, as a result, classical minimization techniques will leave SSM'' unchanged.

[2]The current implementation does not include this algorithm. However, in section 4.4, I describe the rules I used for adding loops to state machines.

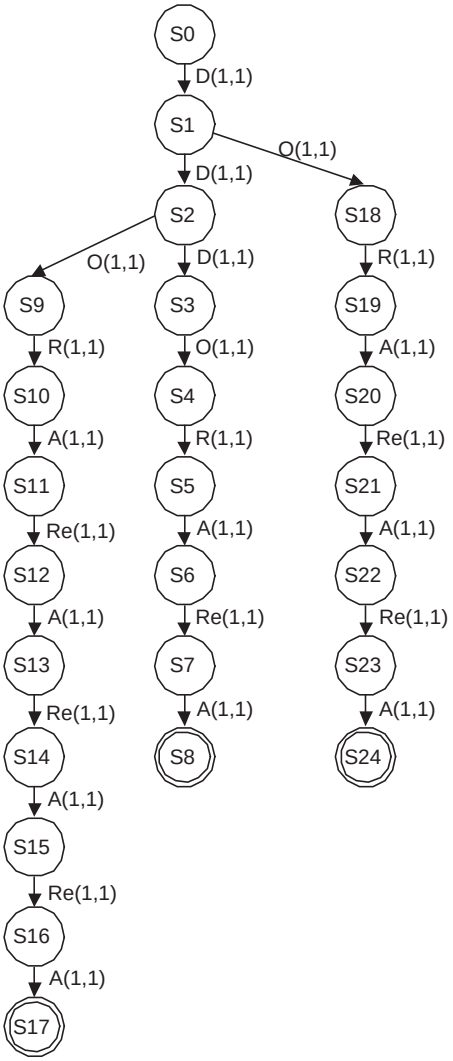

Figure 3.3: The Session State Machine (*SSM*) generated.

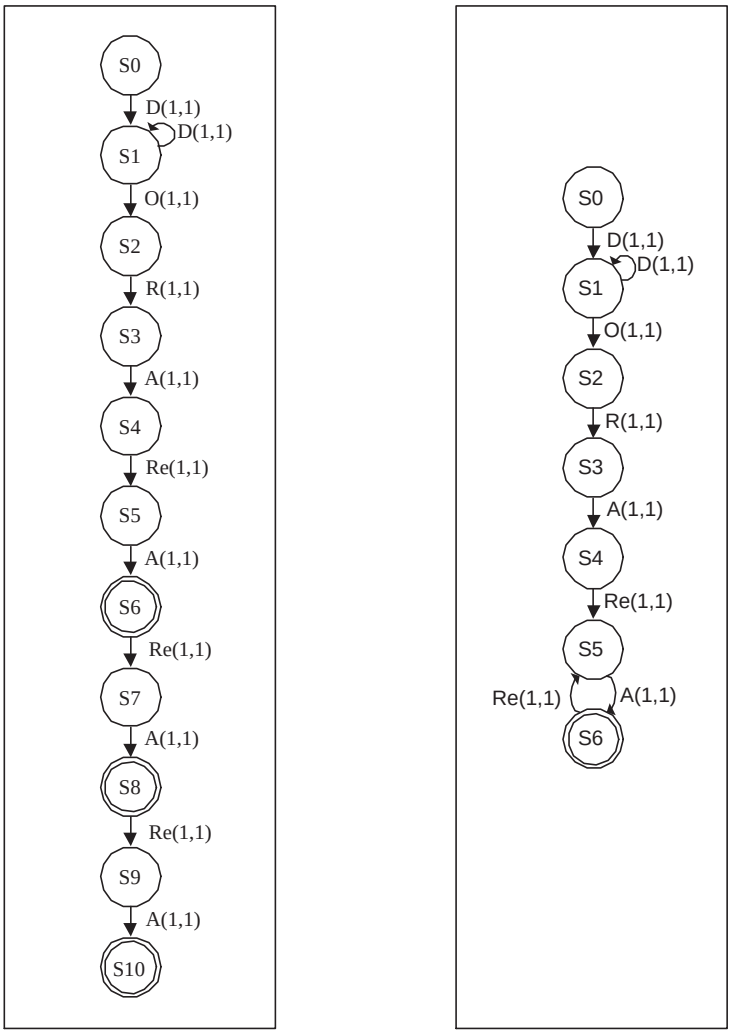

Figure 3.4: The Session State Machine after the addition of loops, after the first and the second iteration of loops algorithm respectively.

3.1.5 From State Machines to TIOA

The final step of the process of creating a formal model of an implementation is the translation of the generated state machine to a formal language, TIOA [22] in our case. The algorithm that does this translation is described in section 5.1. Here, I present the TIOA model that corresponds to state machine SSM'' (figure 3.4).

The *delay* values in this example were added randomly. This is because the messages in the trace presented above had no time stamps. However, in a real example, the actual time stamps are used for calculating the elapsed time between two messages, which corresponds to the time that the model should delay between the execution of two consecutive actions.

Here, I only include one send and one receive actions. The complete model is presented in appendix A.

```
%Input Action
input receive(m) where m.msgType = D ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = 0
  eff
    if(curState = 0) then
       queue := enQ(MKtimed_message(m,clock), queue);
       curState := 1;
       delay := 0;
    fi;

%Output Action
output send(m) where m.msgType = O ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = delay + 5
   pre
      curState = 1;
   eff
      queue := enQ(MKtimed_message(m,clock), queue);
      curState := 2;
      delay := delay + 5;
```

Chapter 4

Building Models from Observations

Creating models represented as state machines from a set of words is a well studied problem, where different approaches have been proposed. Different techniques where words are provided in the form of Regular Expressions or Message Sequence Charts (MSCs) are available [2, 15, 37]. In addition, more theoretical approaches have been proposed. A representative example is Angluin's work presented in [4], where the author assumes the existence of an oracle capable of answering queries for guiding the generation of a model.

This proposed network testing approach has some similarities with these approaches, considering that some of their goals, i.e., the generation of a state machine, are similar. However, because of some different challenges that the proposed network testing approach faces, these approaches cannot be used directly.

First, Angluin's algorithm cannot be used directly due to the lack of an oracle that has the desired properties. More precisely, an oracle as defined by Angluin

is capable of answering membership queries. That is, given a state machine, the oracle either accepts the state machine (i.e., the state machine is complete) or gives a counterexample that violates the state machine, i.e., a sequence of events that should be accepted by the state machine but is not.

Also, approaches based on Regular Expressions or MSCs cannot be used directly. This is because this work assumes that the only realistic approach for network testing is black box testing, and thus the tester can only observe the external behavior of an IUT. By monitoring its external behavior the tester collects a set of observations. Assuming that each observation represents a permissible word, then approaches based on Regular Expressions or MSCs can be used. However, models derived using these approaches will be extremely restrictive, because they will only accept the set of observations collected. In addition, these approaches have a huge overhead for creating deterministic from non-deterministic finite automata.

Because of the limitations mentioned above, a novel approach for building models represented as state machines is proposed. The proposed approach is capable of creating more complete models efficiently and effectively. The steps of this process are as follows:

1. Analyze and break each observed sequence of messages into smaller subsequences, each capturing a permissible behavior of a component (or a collection of components) of the implementation. Each subsequence is treated as a word that the generated model accepts.

2. Use field semantics to abstract each word created in step 1 to allow the creation of a more complete model.

3. Build a state machine using the abstracted words created in step 2.

4. Minimize the state machine created in step 3 by both identifying and adding loops and also performing classical minimization techniques [2].

The following sections describe each step in detail.

4.1 Extracting Words from Observations

Generating an automaton from a set of observations requires making some assumptions about the way components of the IUT use fields. The first assumption is that there are related sequences of messages, hereafter defined as *sessions*. One can think of a session as a permissible behavior of a component of the IUT. Sessions are treated as words for building a model of the IUT. The second assumption is that some of the fields are session identifiers, i.e., fields that can be used for assigning messages to sessions. The final assumption is that the way a component processes a session is similar to the way it processes any other session, e.g. the same sequence of messages might appear during the interaction between a server and any client.

For identifying the session identifier fields, the tester should utilize both the specification of the IUT and the property he/she wants to prove. For example, assume that a tester wants to test the component of a DHCP server that handles the communication between a DHCP server and a DHCP client. Then, the session identifier field for grouping messages together involving the communication between a DHCP server and a DHCP client is the *transaction_ID* field. Another example is the DHCP component that manages the allocation of the IP addresses.

Sessions for creating a model of this component should include all messages involving the assignment of a single IP address. The session identifier includes the offered IP address in DHCP offer or acknowledgement messages from the server; the requested IP address in DHCP request messages from the client; and the client address in DHCP requests to renew a lease on an IP address. Each session represents how a DHCP server allocates a particular IP address to different clients, supposedly one client after another.

A more complicated example is that of a TCP session (a single TCP session transmits a single stream of data between two endpoints). Since TCP is a connection oriented protocol, a special sequence of messages is used for initializing and terminating a session. Thus, the actual values to be used for assigning messages to sessions (i.e., the actual values that source and destination IP address and source and destination port should carry) should be extracted from some special messages (i.e., the session initialization messages). Session identifying fields are the SYN, FIN and RST flag fields, the source and destination IP address fields, and source and destination port fields.

In this dissertation, I investigated two different algorithms for creating sessions. The first one is easier and faster but also more restrictive in terms of the sessions it can create. It groups messages together using equality on fields. For example in a DHCP communication session, two messages belong to the same group if the predicate $(message_1.transaction\_ID == message_2.transaction\_ID)$ is true, i.e., the value carried by field $transaction\_ID$ in $message_1$ is equal to the value carried by the same field in $message_2$.

However, there are cases where the above approach is not sufficient. One example is the TCP session described above. Also, examples where the same

message might belong to multiple sessions cannot be handled by this approach. For example, in TCP, an acknowledgment message might acknowledge more than one messages and thus belong to multiple sessions.

Due to the above limitations, I investigated a different, more general approach for grouping messages together. The goal was to allow the creation of any arbitrary group. The only restriction that applies is to maintain the time ordering, i.e., if m_1 precedes m_2 in the trace, then m_1 should also precede m_2 in a session.

The idea I adopted was to load the trace into a database and create sessions based on the results of a new, extended form of a relation algebra query, the *Parameterized Query*. *Parameterized Query* is a relational algebra query extended with parameters. By instantiating the parameters of a parameterized query, it becomes an ordinary relational algebra query. All tuples returned by a single instantiated parameterized query represent a session. A formal definition of *Parameterized Queries* along with optimization used for efficiently calculating sessions using those is presented in chapter 6.

4.2 Field Semantics

In order to facilitate the creation of more general models, messages in sessions can be abstracted. This is achieved by assigning different semantics to each field in a message. More precisely, three types of fields based on their semantics are defined: *Symbolic*, *Regular* and *Invisible*.

Fields whose values are arbitrary, with no internal significance, such that different values will be treated similarly by network components are called *symbolic fields*. For example, a requested IP address is an arbitrary identifier in DHCP.

Clients and servers treat the IP address in the same way, regardless of its value. Similarly, the offered IP address is an arbitrary identifier. The server ID is an arbitrary identifier to the client; the client hardware address is an arbitrary identifier to the server.

For fields whose values are arbitrary, one can assume that any permutation of values of the symbolic fields that has the same pattern as a permutation of values of the same fields in a legal session results in another legal session. As a result, actual values of symbolic fields are replaced by symbolic values calculated as follows:

Definition 4.2.1. Given any sequence of values $x_1, ..., x_k$, define the symbolic representation $n_1, ..., n_k$ of the sequence $x_1, ..., x_k$ as follows:

1. $n_1 = 1$.

2. $n_i = n_j$ if there is a $j < i$ such that $x_i = x_j$.

3. $n_i = size(\{x_j \mid j < i \text{ AND } x_j \neq x_i\})$ otherwise.

The above definition says that the first time a value appears in a sequence, its value in the symbolic representation is equal to the number of distinct values appeared before it in the sequence. For example, the symbolic representation of the sequence a, a, d, a, b, c, a, b, d is 1, 1, 2, 1, 3, 4, 1, 3, 2. Note that the first value in the symbolic representation of any sequence is 1.

In addition to ordinary symbolic fields defined above, two more types of symbolic fields have been defined, *Ordered-Symbolic* and *Symbolic-Group* fields. This is because there are cases where ordinary symbolic fields are insufficient for capturing some desirable behaviors.

The first limitation of ordinary symbolic fields is that they do not take into consideration any information that a session might contain because of the order in which some values appear in a session. For example, assume that the tester considers it valuable to know that packets arrive in order in a TCP session, i.e., values carried by the sequence fields of packets arrived are monotonically increasing. However, this is impossible when using ordinary symbolic representation, because the value carried by the first message arrived will always be mapped to symbolic value 1, the second to 2 and so on, despite the actual value they carry. To solve this problem, a new field type is defined that extends the symbolic field type by adding an ordering function. Before assigning symbolic values to fields, an ordering function orders messages based on their actual value. For each different *Ordered-Symbolic* field a different ordering function is created. Given an ordered sequence of messages, the assignment of ordered symbolic values is similar to the assignment of symbolic values when ordinary symbolic fields are used. This is described in definition 4.2.2.

Definition 4.2.2. An Ordering Function O on an Ordered-Symbolic field f, denoted as $O(f)$, is a function that, given a session $S = m_1, ..., m_n$, orders $m_1(f)$, ..., $m_n(f)$ such that for every $i, j \in n$ and $i \neq j$, $O(m_i(f)) < O(m_j(f))$ if and only if $m_i(f) < m_j(f)$.

A second limitation of ordinary symbolic fields is that symbolic values of two different fields are not related, i.e., one cannot extract any information by comparing the symbolic values of different fields. However, there are cases where this is desirable. To make this clear consider the following example: It is known that in TCP there is a relation between the values carried by sequence fields and the values carried by acknowledgment fields in the same session. If the user decides

to define those two fields as symbolic, then there will be no relation between the symbolic values assigned to these fields. This is because different pools are used for assigning symbolic values to each field. In cases where the user decides that relations between symbolic values of different fields might be useful in understanding the functionality and analyzing a protocol, then he/she can define groups of symbolic fields. In a *Symbolic-Group*, the symbolic values of all of its members are assigned from the same pool. As a result, the symbolic values of different fields that belong to the same group are related.

The combination of the two extensions of ordinary symbolic fields, *Ordered-Symbolic* and *Symbolic-Group*, is also legal. To allow this, the *Ordered-Symbolic-Group* field was defined. Fields that belong to an *Ordered-Symbolic-Group* are both grouped and ordered.

In addition, the tester is allowed to restrict symbolic groups based on the direction (input or output) of a message. For example, in TCP, the *sequence* field of outgoing messages should be grouped with the *acknowledgment* field of incoming messages. For this reason the *Input(m)* and *Output(m)* predicates are defined. The former returns true if the message m is an incoming message, and the latter returns true if m is an outgoing message. Examples of how one can define symbolic fields of any type as well as how groups can be restricted based on the direction of messages are presented in section 4.2.

The second semantic that one can assign to a field is *Regular*. *Regular fields* are fields whose actual value is significant for state transition in the generated state machine. As a result, the regular fields remain unchanged. For example, assume that a DHCP client is at a state waiting for acknowledgment or negative acknowledgment response from the server. In this case, it is significant for the

client to know the actual value of the field carrying the server's response, since different values of that field will trigger transitions to different states.

Finally, any remaining field, i.e., fields defined in the specification and not assigned to any of the above categories, are called *Invisible Fields*. These are fields of no importance to the state transition and will remain invisible to both the program and the user.

Definition 4.2.3. Given a message M, define the *Normal Representation* (NR) of M to be a message M' that contains only the actual values of the regular fields and the symbolic representation of the symbolic fields of M.

Equality of messages using the above semantics is defined as follows:

Definition 4.2.4. Let m_1 and m_2 be two messages. Denote $NR(m)$ as the function that returns the normal representation of m. Then m_1 is equivalent to m_2 if $NR(m_1) = NR(m_2)$.

Based on definition 4.2.4, two messages are equal if the actual values of their regular fields and the symbolic values of their symbolic fields are equal.

4.3 Session State Machine

This section describes the algorithm that creates a model represented as a state machine from a set of session, the *Session State Machine*. A *Session State Machine* recognizes every session defined from the network trace, in addition to others. Equivalent sessions follow the same paths through the graph representing the state machine.

Definition 4.3.1. Let $m_1, ..., m_k$ and $w_1, ..., w_k$ be two sequences of messages (i.e., sessions). Denote the value of a field f of a message by $m(f)$. Then $m_1, ..., m_k$ is **equivalent** to $w_1, ..., w_k$ if:

1. For each regular field r, $m_i(r) = w_i(r)$ for all i.

2. For each symbolic field p, there is a *permutation* π_p of the values of p such that $\pi_p(m_i(p)) = w_i(p)$ for all i.

Definition 4.3.2. Two sessions $m_1, ..., m_n$ and $w_1, ..., w_m$ are *k-equivalent* if there exists a $k \leq \min(n, m)$ such that for all $i \leq k$ m_i is equivalent to w_i.

Lemma 4.3.3 justifies the use of the symbolic representation in place of the actual values, by stating that if two sequences of values over a domain are related by a permutation of the values in the domain, then their symbolic representation is the same, and vice versa. Note that a permutation over a set of values S is a bijection on S, that is, a mapping $f : S \rightarrow S$ that is one-to-one and onto.

Lemma 4.3.3. *Let $X = x_1, ..., x_k$ and $Y = y_1, ..., y_k$ be sequences of values taken from a domain D. Then there is a permutation π of D such the $\pi(x_i) = y_i$ for all i if and only if the symbolic representation for X equals the symbolic representation for Y.*

Proof. Let $n_1, ..., n_k$ be the symbolic representation of $x_1, ..., x_k$ and $m_1, ..., m_k$ be the symbolic representation of $y_1, ..., y_k$.

First, if there is a permutation π from X to Y, such that $\pi(x_i) = y_i$, then $y_i = y_j$ if and only if $x_i = x_j$. Using induction, suppose that the symbolic representations of the sequences are the same up to an index k_0. Then if $x_{k0+1} = x_i$ for some $i < k_0$, it must also be true that $y_{k0+1} = y_i$. Thus $n_{k0+1} = n_i$ and

$m_{k0+1} = m_i$, but by induction hypothesis $n_i = m_i$. Otherwise, suppose that x_{k0+1} $\neq x_i$ for any $i < k_0$. By the induction hypothesis, the number of distinct values in $x_1, ..., x_{k0}$ is the same as the number of distinct values in $y_1, ..., y_{k0}$, and therefore $n_{k0+1} = m_{k0+1}$.

Conversely, assume that for all *i = 1, 2, ..., k*, $n_i = m_i$. Again using induction, assume that there is a permutation π satisfying $\pi(x_i) = y_i$ for *i = 1, 2, ..., k*$_0$. If $x_{k0+1} = x_i$ for some $i < k_0 + 1$, then $n_{k0+1} = n_i$ and therefore $m_{k0+1} = m_i$. But this implies that $y_{k0+1} = y_i$, so the same permutation works for *i = 1, 2, ..., k*$_{0+1}$. If there is no $i < k_0 + 1$ with $x_{k0+1} = x_i$, then $n_i > n_j$ for all $j < i$ and hence $m_i > m_j$ for all $j < i$. Thus one can conclude that there is also no $i < k_0 + 1$ such that $y_{k0+1} = y_i$. Composing π with the permutation that swaps $\pi(x_{k0+1})$ with y_{k0+1} gives the desired permutation. □

A second lemma says that one can determine the equivalence of sequences of messages efficiently using the symbolic representations.

Lemma 4.3.4. *Two sequences of messages $m_1, ..., m_n$ and $w_1, ..., w_n$ are equivalent if and only if the symbolic representation of symbolic fields as well as the actual value of regular fields of corresponding messages are equal.*

Proof. Follows immediately by definition 4.3.1 and lemma 4.3.3. □

A Session State Machine is constructed so that there is a path p_1, ..., p_k from the initial state (root) to an accepting state (leaf) for any session S=m_1, ...,m_k used for building the state machine. This path has the property that the normal representation of m_i is the label of edge p_i. The tree has the property that two sessions follow the same path for k steps if and only if the sequences of messages

consisting of the first k messages in each session are equivalent, i.e., they are *k-equivalent*.

The algorithm uses the normal representation of a message, i.e., a message that contains only the actual values of the regular fields and the symbolic values of the symbolic fields. $NR(m)$ denotes the normal representation of a message m.

1. The first step is to initialize the tree with a single start state (the root of the tree).

2. Add the first session to the tree. Suppose the first session has messages m_1, ..., m_n. Then, after adding the first session, the tree has a single path starting at the root. The edge below the root is labeled $NR(m_1)$; the next edge on the path is labeled $NR(m_2)$; and so on, to the edge above the leaf, which is labeled $NR(m_n)$.

3. Repeat for each session: To process each subsequent session w_1, ..., w_k in the trace, find the path with the largest k such that $NR(w_1)$, ..., $NR(w_k)$ is the same as $NR(m_1)$, ..., $NR(m_k)$.

   (a) w_1. Choosing the first edge requires finding the edge below the root such that $NR(m_1) = NR(w_1)$. If there is no message m_1 on an edge below the root with corresponding regular and symbolic fields being equal, then create a new edge below the root with label containing $NR(w_1)$ and having a new state at its child end. The remaining messages $NR(w_2)$, ..., $NR(w_n)$ will be the edge labels for a new path below the edge $NR(w_1)$.

   (b) w_{h+1}. Assume that after processing the first h messages, the sequence of messages w_1, ..., w_h follows a path leading to state s_h in the tree.

If there is an edge whose label is $NR(m_{h+1})$ that has the same values for regular and symbolic fields as w_{h+1}, then the sequence of messages w_1, ..., w_{h+1} leads to state s_{h+1} at the child end of the edge. If there is no such edge, add a new edge below s_h and use $NR(w_{h+1})$ as its label. The node at the child end of the edge is a new state.

Theorem 4.3.5. *Two sessions, m_1, ..., m_k and w_1, ..., w_k lead to the same node at level k of the tree (i.e., to the same state) if and only if they are* k-equivalent.

Proof. By induction on k. □

In addition to the normal representation of a message, edges of a Session State Machine also carry information related to time. More precisely, they carry a list of possible times that the edge can be traversed. This list is called the *time list*. An entry in the *time list* corresponds to the processed time stamp of a message. Each time a session traverses an edge, the processed time stamp of the corresponding message is added to the *time list* of the edge. Considering that many k-equivalent sessions might exist, the *time list* of an edge might contain multiple entries.

Two different approaches have been adopted for processing the time stamp of different messages, *root time reset* and *sink time reset*. The former approach assumes that the first message in a session happens at time 0, i.e., its time stamp is set to 0. For each subsequent message, its time stamp will be equal to the time difference of the current from the previous message in the session. Using this approach, all edges going out of the root node have time stamps equal 0.

There are protocols where the actual time that a session starts is when the last message of the last session was sent and not when the first message of the new session was observed. For example, in Spanning Tree Protocol (STP) [31]

a session (called round in STP) ends when a switch sends a message to all its neighbors that, among others, contains information regarding the root bridge. At that point, a new round starts, during which the switch collects information for recalculating the root bridge. If the tester decides to build a model where each session corresponds to a round, then the actual time that a session starts is the time that the last message of the previous round was sent. In that case, the latter approach should be used, where the time stamp of the first message in a session will be the time elapsed since the last message of the last session was sent. Time stamps of all other messages are calculated as in the first approach, i.e., the time difference of the current from the previous message.

4.4 Adding Loops to a Model

Session State Machines created by the algorithm presented in previous sections are trees, where a path from the root to a leaf represents a word [1] that corresponds to an abstracted behavior of the IUT. In this section, a beginning of an investigation of how loops can be identified in a set of words is provided. First, in subsection 4.4.1, a formal definition of loops is provided, along with the necessary theory that justifies the definition. Subsection 4.4.2 presents an algorithm that automatically identifies and adds loops to a Session State Machine.

4.4.1 Definitions

Adding loops to Session State Machines requires the definition of some heuristics that will be used as rules for identifying loops that exist in the state machine.

[1] Terms word and session are used interchangeably.

One possible heuristic is to identify as a loop any subsequence in a session that is repeated more than once. However, this is a very general rule and thus many incorrect loops may be identified. A more restrictive rule is to require a potential loop to appear in two different words and the number of times the loop is repeated in each word is not the same. This rule is more restrictive and thus identifies loops more accurately. However, by only using two words, there is a scenario that I call *silly loop syndrome*, where wrong loops can be identified. This syndrome appears when the head and the tail of a loop are the same. Assume a set of words created from the regular expression $(ababa)^*$, i.e., $L = \{W_1 = ababa, W_2 = abababaababa, W_3 = abababaababaababa, ...\}$. Also assume that W_1 and W_2 are used for identifying and adding loops to W_2. This pair of words can create three possible loops $abab(a)^*$, $ab(aba)^*$ and $(ababa)^*$. However, given L, the choice that has the highest probability to be correct is the last one. The next theorem claims that the *silly loops syndrome* does not exists in the case that a loop appears in three words and the number of times it is repeated in each word is different.

Theorem 4.4.1. *Assume $L = \{W_1, ..., W_p\}$ and a loop X. The silly loops syndrome does not exists if three words W_i,W_j,W_k, where $W_i = PX^lS_i$, $W_j = PX^mS_j$, $W_k = PX^nS_k$, and $l \neq m \neq n$ and $k, m, n > 0$ are used for identifying a loop.*

Proof. Assume three words $W_1 = PXS_1$, $W_2 = PXXS_2$, $W_3 = PXXXS_3$, where P is a prefix, S is a suffix and X is the loop. For the *silly loop syndrome* to occur, $X = Y^mQY^n$. This implies that $W_1 = PY^mQY^nS_1$, $W_2 = PY^mQY^nY^mQY^nS_2$, $W_3 = PY^mQY^nY^mQY^nY^mQY^nS_3$.

Based on the above, Y will be considered to be a loop only if it is repeated for a different number of times in those three words. This can only happen

if $Y \in head(Q)$. By substituting RY for Q one gets $W_1 = PY^mRYY^nS_1$, $W_2 = PY^mRYY^nY^mRYY^nS_2$, $W_3 = PY^mRYY^nY^mRYY^nY^mRYY^nS_3$. Proceeding recursively, X will eventually become equal to Y^k which contradicts the assumption made above. This is illustrated in Figure 4.1.

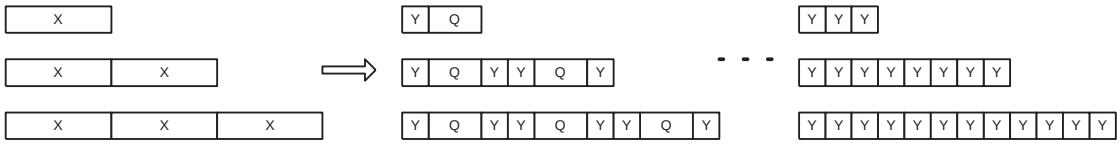

Figure 4.1: Silly Loop proof.

□

In what follows, I formally define loops. This definition can be used as a rule for identifying loops in state machines. For simplicity and clarity, loops are defined in terms of the words (sessions) used for building the Session State Machine. In addition, the term *regular expression* is used to describe words after loops are added. However, I first start with an auxiliary definition.

Definition 4.4.2. For any word W, $head(W) = \{Y | W = YX$ for some word X$\}$. Similarly, $tail(W) = \{Y | W = XY$ for some word X$\}$.

Next, I formally define loops.

Definition 4.4.3. Given a set of words $L = \{W_1, W_2, ..., W_k\}$, define a loop to be any subsequence X that appears in at least three words $W_i, W_j, W_k \in L$, $W_i = PS_i$, $W_j = PX^mS_j$, $W_k = PX^nS_k$, where P is the longest common prefix of W_i, W_j, W_k, and $X^l \in tail(P)$, $l \geq 1$, $m \neq n$ and $m, n > 0$, $X \neq Y^t$ for any nonempty string Y and $t \geq 2$, and S_i, S_j, S_k are suffixes such that $head(S_i), head(S_j), head(S_k)$ do not contain X. In addition, all loops that are

in P have already been identified, i.e., loops should be introduced starting from the beginning of each word.

In what follows, I provide some lemmas that prove some properties of definition 4.4.3, which assist in giving special consideration to overlapping loops, considering that these can be a source of ambiguity. This is because only one of the overlapping loops should be used. Overlapping loops are illustrated in figure 4.2. For this example, assume that $W = PX^mS = QY^nT$, where W is a word, P and Q are prefixes, S and T are suffixes and X and Y are the overlapping loops. As a result, the approach should unambiguously decide which loop (X or Y) to introduce to W.

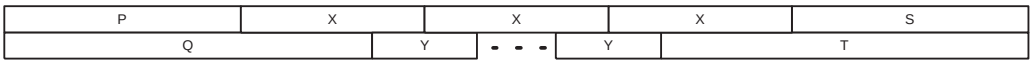

Figure 4.2: Overlapping loops appearing in a single word W.

Definition 4.4.4. For a word W containing a subsequence X that is a loop (as defined in 4.4.3), define $loop(W, X)$=PX^*S for P and S such that W=PX^iS for some i and $P \neq QX$ for any Q and $S \neq XT$ for any T.

Theorem 4.4.5. *If X and Y are loops of the same length, both appearing as subsequences of a word W, and if the first appearance of X in W starts before the first appearance of Y in W, then $loop(W, X)$ contains $loop(W, Y)$.*

Proof. Since $|X| = |Y|$ and X and Y are overlapping, one can set $X = W_2W_1$ and $Y = W_1W_2$ as illustrated in Figures 4.3 and 4.4. Assuming (without loss of generality) that Y is the loop that starts first, two cases should be considered. First, when the $head(T) = head(Y)$ and second, when $head(S) = tail(Y)$

(illustrated in Figures 4.3 and 4.4 respectively). In the first case, using X as a loop, W can be simplified to $QW_1(W_2W_1)^*S$ and using Y as a loop, W will be equal to $Q(W_1W_2)^*W_1S$, which are equivalent. As a result, in this case one can chose any one of the loops, since they are equivalent. For the second case, i.e., $head(S) = tail(Y)$, using X as a loop W becomes $W = QW_1(W_2W_1)^*W_2T$ and Y as a loop W' becomes $W' = Q(W_1W_2)^*S$. In this case, $W \subseteq W'$, so W' which is created by the loop that starts first should be preferred.

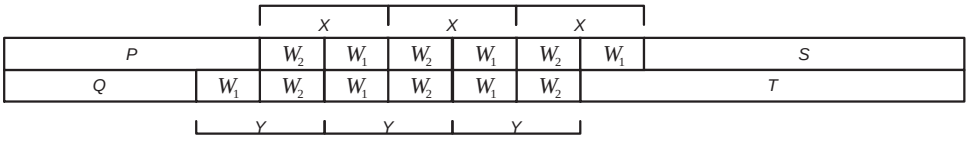

Figure 4.3: Overlapping Loops when $|X| = |Y|$ and $head(T) = head(Y)$.

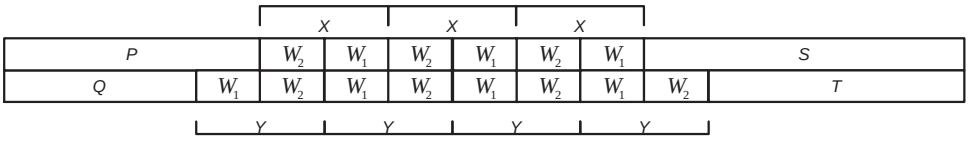

Figure 4.4: Overlapping Loops when $|X| = |Y|$ and $head(S) = tail(Y)$.

□

Theorem 4.4.5 suggests that if two overlapping loops have the same size, then the one that starts first creates a regular expression that is more complete, i.e., it can create more words. Since the overall goal of the proposed approach that creates models from observation is to create a model that is as complete as possible, loops that start first should be preferred. This is the reason that definition 4.4.3 requires that the prefix P of a word W should contain all possible loops before adding any loop after P. In what follows, I show that a single word W cannot

have two loops X and Y, where $|X| < |Y|$ and $YYYYY...Y$ (i.e., Y^n) overlaps with three X (i.e., XXX). This is why definition 4.4.3 requires that a loop is repeated three times. However, before proving that, I prove an auxiliary lemma necessary for the proof.

Lemma 4.4.6. *Given two words X_1 and X_2, then X_1X_2=X_2X_1 is true if and only if $X_1 = Y^m$ and $X_2 = Y^n$ for m,n $\geq$ 1.*

Proof. Clearly the one direction is obvious. If $X_1 = Y^m$ and $X_2 = Y^n$ then $X_1X_2 = Y^mY^n = Y^{m+n}$ and $X_2X_1 = Y^nY^m = Y^{n+m}$.

For proving the other direction, the proof was split into three cases based on the size of X_1 and X_2. The first case is when X_1 and X_2 have the same size, i.e., $|X_1| = |X_2|$. In that case, clearly $X_1 = X_2$ (Figure 4.5). As a result, $X_1 = X_2 = Y$ and $m = n = 1$.

X_1	X_2
X_2	X_1

Figure 4.5: X_1X_2=X_2X_1 and $|X_1| = |X_2|$

The other two cases, i.e., $|X_1| > |X_2|$ and $|X_1| < |X_2|$, are symmetric and will be proved together using induction on the size of the shortest word. Without loss of generality, assume that $|X_1| > |X_2|$. For proving the base case, assume that $|X_2|$=1 and $X_2 = a$. However, for proving this base case, induction on the size of X_1 should be used. For the base case of this "sub-induction" and since $|X_1| > |X_2|$, one should assume that $|X_1|$=2. Since X_2=a, one can replace X_1 with aX_3 (Figure 4.6), which results in X_3a=aX_3. Using the assumption that $|X_1|$=2 and $X_1 = X_3a$=aX_3, one can conclude that $X_1 = a^2$. As an induction hypothesis

of the "sub-induction", assume that if $|X_1| = n - 1$, $X_1 = a^{n-1}$. If $|X_1| = n$ and $X_1 = X_3a{=}aX_3$, $|X_3| = n-1$, and based on the induction hypothesis, $X_3 = a^{n-1}$. This implies that $X_1 = a^n$, which finishes the proof of the "sub-induction" and also the base case of the theorem.

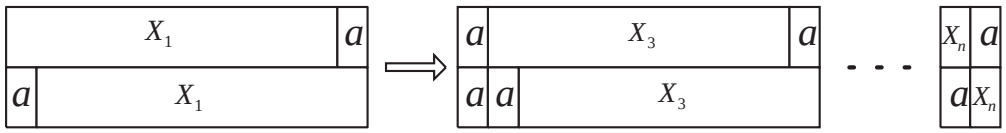

Figure 4.6: Induction when $|X_2|$=1.

For the induction hypothesis of the main theorem, one can assume that theorem 4.4.6 is correct for any size of X_2 smaller that n. Assuming that, the correctness of the theorem in the case that $|X_2| = n + 1$ should be proved. Given that $|X_1| > |X_2|$ one can replace X_1 with X_2X_3 (Figure 4.7). As before, the proof should be split into three cases based on the size of X_2 and X_3. If they are equal in size, then $X_2 = X_3 = Y$, which implies that $X_1 = X_2X_3 = YY$, which proves this case. The other two cases, i.e., $|X_2| < |X_3|$ or $|X_2| > |X_3|$, are symmetric and will be proved together assuming, without loss of generality, that $|X_2| > |X_3|$. By repeatedly substituting the longer with shorter words, eventually one should either find a substring which is equal to X_2, which will prove that $X_1 = Y^m$ and $X_2 = Y^n$, or a substring $|X_k| < |X_2|$, where $X_{k-1} = X_2X_k$, $X_{k-2} = X_2X_{k-1}$, ..., $X_1 = X_2X_3$. In the latter case, one can use the inductive hypothesis and the fact that X_k is the smallest of the two words, as well as the fact that $|X_k| \leq n$ to complete the proof.

□

Lemma 4.4.7 states that it is impossible to have a word that has two loops X and Y, where $|X| > |Y|$ and also $X \neq Z^m$ and $Y \neq W^n$ (i.e., X and Y are the

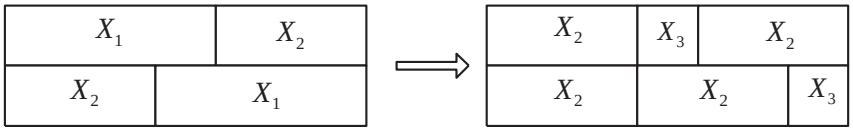

Figure 4.7: $|X_1| > |X_2|$ and replacement of X_1 with X_2X_3.

smallest possible loops) overlapping with each other if Y^k overlaps with at least three X.

Lemma 4.4.7. *A single word W cannot have two loops X and Y, where $|X| > |Y|$, $X \neq Z^m$, and $Y \neq R^n$, such that Y^k overlaps with at least three X.*

Proof. Assume that $Y = Y_1Y_2 = Y_3Y_4$ (Figure 4.8). Y_1 is part of Y that intersects with the tail of the first X. Similarly, Y_2 is the part of Y that intersects with the head of the second X, Y_3 is the part of Y that intersects with the tail of the second X and finally Y_4 is the part of Y that intersects with the head of the third X.

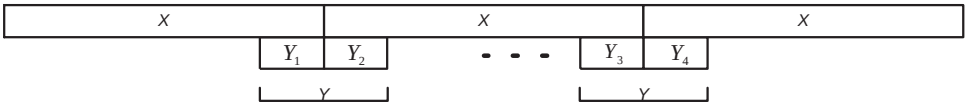

Figure 4.8: Split Y into sub-words, where $Y = Y_1Y_2 = Y_3Y_4$.

This proof starts with the two cases: first, X ends with Y, which also implies that the second X starts with Y (Figure 4.10); and, second, X ends with Y, which also implies that the third X starts with Y (Figure 4.9). These cases indicate that at some repetition of X and Y, they either start or end at the same position. For the former case (the first X ends with Y), one can conclude that $Y = Y_4Y_3$ (using the tails of the first and second x). This implies that $Y = Y_3Y_4 = Y_4Y_3$. Also from theorem 4.4.6 one can conclude that $Y_3 = Z^m$ and $Y_4 = Z^n$. As a result, both X and Y have inner loops, which contradicts one of the assumption of theorem

4.4.7. Similarly, for the latter case, $Y = Y_2Y_1$ (using the heads of the second and third X). This implies that $Y = Y_1Y_2 = Y_2Y_1$ and from theorem 4.4.6 one can conclude that both X and Y have inner loops, which is a contradiction.

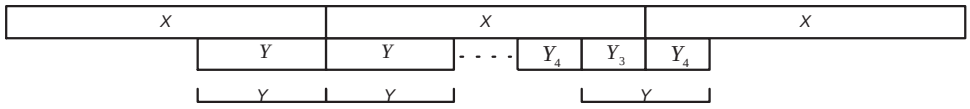

Figure 4.9: Y loop ends where the first X loop ends.

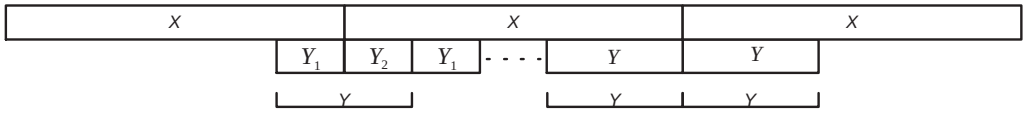

Figure 4.10: Y loop starts where the third X loop starts.

Next, one should consider the case that X and Y do not start or end at the same position. Similarly to previous proofs, one must consider all possible cases based on the sizes of Y_1, Y_2, Y_3, and Y_4. First, consider the case that $|Y_1| = |Y_3|$ and $|Y_2| = |Y_4|$. In this case, trivially $X = (Y_2Y_1)^m$, which is a contradiction since it violates the assumptions that $X \neq Z^m$.

The other two cases are symmetric and without loss of generality, assume that $|Y_2| < |Y_4|$ (which also implies that $|Y_1| > |Y_3|$ since Y=Y_1Y_2=Y_3Y_4). Having that in mind and using Y, one can imply that $Y_1 = Y_3Y_6$ and $Y_4 = Y_6Y_2$ (Figure 4.11 A). Similarly, using the head of X, one can imply that $Y_1 = Y_5Y_3$ and $Y_4 = Y_2Y_5$ (Figure 4.11 B).

Based on the above one can extract the following information:

1. $|Y_5| = |Y_6|$.

2. $Y = Y_1Y_2 = Y_3Y_4$.

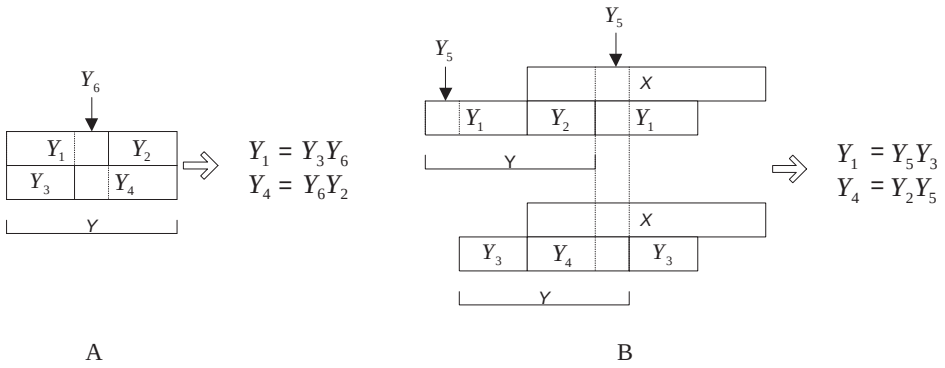

Figure 4.11: Using Y and the head of X to simplify Y_1 and Y_4.

3. $Y_2Y_1 = Y_4Y_3$.

4. $Y_1 = Y_3Y_6 = Y_5Y_3$.

5. $Y_4 = Y_6Y_2 = Y_2Y_5$.

Using 4 and 5, the only way for 1 - 5 to hold is when either X or Y contain smaller loops, which contradicts the assumption that $X \neq Z^m$ and $Y \neq W^n$. As before, the proof is split into different cases based on the relative sizes of Y_2, Y_3, Y_5 and Y_6. First, assume that the size of Y_5 and Y_6 is the same as the size of either Y_2 or Y_3. Without loss of generality, also assume that $|Y_5|$,$|Y_6|$=$|Y_2|$. In that case, using 5, one can conclude that Y_5=Y_6=Y_2. Since Y_5=Y_6 and by substituting Y_6 with Y_5 in 4, one gets $Y_3Y_5 = Y_5Y_3$. From theorem 4.4.6 one can conclude that $Y_3 = W^m$ and $Y_5 = W^n$. As a result, $Y = Y_1Y_2 = Y_3Y_6Y_2 = W^mW^nW^n = W^{m+n+n}$, which is a contradiction.

For proving the case where neither $|Y_2|$ nor $|Y_3|$ is equal to $|Y_5|$,$|Y_6|$, I use induction on the size of the shortest word. This creates two cases, $|Y_5|$,$|Y_6| < \max(|Y_2|, |Y_3|)$, i.e., the size of at least one of Y_2,Y_3 is bigger than the size of Y_5,Y_6 and $|Y_5|$,$|Y_6| > \max(|Y_2|, |Y_3|)$, i.e., the size of Y_5,Y_6 is bigger than the size of both

Y_2 and Y_3. I start the proof by proving the base case for both cases. First, assume that $|Y_5|$,$|Y_6| < \max(|Y_2|, |Y_3|)$ and that $|Y_2| \geq |Y_3|$. For the base case, I assume that $|Y_5|$=$|Y_6|$=1. Again, this should be split into two cases, when Y_5=Y_6 and when $Y_5 \neq Y_6$. However, the former case is trivial, since if Y_5=Y_6 and utilizing theorem 4.4.6, one can prove that $Y = W^n$, regardless of the size of Y_2 and Y_3. As a result, I concentrate on the latter case where $Y_5 \neq Y_6$.

Let $Y_5 = a$ and $Y_6 = b$. In this case, using 5, one gets that $Y_4 = Y_6Y_2 = Y_2Y_5 = bY_2 = Y_2a$. However, this implies that $Y_2 = bY_7a$. By substituting Y_2 in 5 one gets $Y_4 = bY_2 = Y_2a = bbY_7a = bY_7aa \Rightarrow bY_7 = Y_7a$. Proceeding recursively, eventually a word Y_n, where $|Y_n| = 1$, is created. In this case, $bY_n = Y_na$, which implies that $Y_n = a = b$, which is a contradiction. Figure 4.12 illustrates this induction.

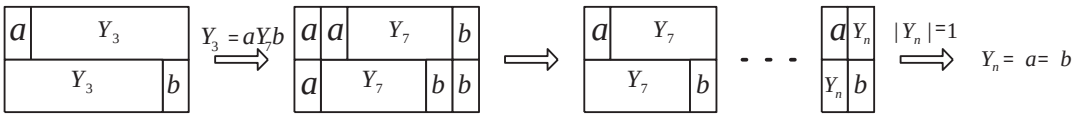

Figure 4.12: Base case when $|Y_5|, |Y_6| < |Y_2|$.

Next, I prove the base case when $|Y_5|$,$|Y_6| > \max(|Y_2|, |Y_3|)$. Again, the proof should be split into two cases, one for $Y_2 = Y_3$ and one for $Y_2 \neq Y_3$. For the base case of the former case, assume that $Y_2 = Y_3 = a$. Replacing Y_2 and Y_3 with a in 4 and 5, they become $Y_1 = aY_6 = Y_5a$ and $Y_4 = Y_6a = aY_5$ respectively. $aY_6 = Y_5a$ implies that Y_5 starts with a and Y_6 ends with a. Similarly, $Y_6a = aY_5$ implies that Y_5 ends with a and Y_6 starts with a. As a result, $Y_5 = aY_7a$ and $Y_6 = aY_8a$. This implies that $Y_1 = aaY_8a = aY_7aa$ and $Y_4 = aY_8aa = aaY_7a$. By simplifying these two equations, they become $aY_8 = Y_7a$ and $Y_8a = aY_7$ perspectively. Proceed recursively until finding Y_n, Y_{n+1} with size equal to one. In that case $Y_n = Y_{n+1} = a$. As a result, $Y_1 = Y_4 = a^k$ and since $Y_2 = a$,

$Y = Y_1Y_2 = a^{k+1}$, which is a contradiction.

Now, assume that $Y_2 = a$ and $Y_3 = b$ ($Y_2 \neq Y_3$). As before, by substituting Y_2 with a and Y_3 with b in 4 and 5 they become $Y_1 = bY_6 = Y_5b$ and $Y_4 = Y_6a = aY_5$ respectively. $bY_6 = Y_5b$ implies that Y_5 starts with b and Y_6 ends with b and $Y_6a = aY_5$ implies that Y_5 ends with a and Y_6 starts with a. As a result, $Y_5 = bY_7a$ and $Y_6 = aY_8b$. This implies that $Y_1 = baY_8b = bY_7ab$ and $Y_4 = aY_8ba = abY_7a$. In this case, $Y = Y_1Y_2 = baY_8ba = bY_7aba$. The only way for this equation to be true is if $Y_7 = (ab)^k$ and $Y_8 = (ba)^k$. However, this implies that $Y = (ba)^{k+2}$, which is a contradiction.

For the induction hypothesis, assume that theorem 4.4.7 holds when max($|Y_2|$, $|Y_3|$, $|Y_5|$, $|Y_6|$) $\leq n$. First, assume that max($|Y_2|, |Y_3|$) $< |Y_5|, |Y_6|$. Since $Y_1 = Y_3Y_6 = Y_5Y_3$ $Y_4 = Y_6Y_2 = Y_2Y_5$ and also max($|Y_2|, |Y_3|$) $\leq |Y_5|, |Y_6|$, one can conclude that Y_5 starts with Y_3 and ends with Y_2. Similarly Y_6 starts with Y_2 and ends with Y_3 (Figure 4.13). As a result, one can set $Y_5 = Y_3Y_7Y_2$ and $Y_6 = Y_2Y_8Y_3$. By substituting the new values of Y_5 and Y_6 one gets $Y_1 = Y_3Y_2Y_8Y_3 = Y_3Y_7Y_2Y_3$ $Y_4 = Y_2Y_8Y_3Y_2 = Y_2Y_3Y_7Y_2$, which implies that $Y_2Y_8 = Y_7Y_2$ $Y_8Y_3 = Y_3Y_7$. The new equations are very similar to the original ones, with the only difference being that their length is smaller. By proceeding recursively, one eventually gets two equations $Y_2Y_{n+1} = Y_nY_2$ $Y_{n+1}Y_3 = Y_3Y_n$, where $|Y_n| < n$, which is true based on the inductive hypothesis. Similarly, one can prove that max($|Y_2|, |Y_3|$) $> |Y_5|, |Y_6|$.

□

Theorem 4.4.5 and lemma 4.4.7 explain why loops should be introduced starting from the beginning of a word if the goal is to create a regular expression that is as complete as possible. They also explain why loops should be repeated at least three times. The final lemma states that a loop belongs to the tail of the

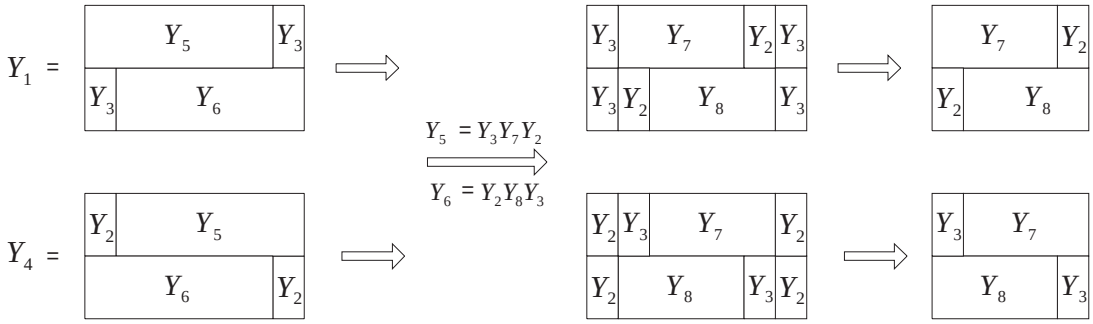

Figure 4.13: Graphic representation of induction.

longest common prefix of the words used for identifying it. This lemma can be used for restricting the positions where a repetition of a loop, which is utilized by the rule for the identification of the loop, can appear.

Lemma 4.4.8. *For any three words W_1, W_2, W_3, let P be their longest common prefix. Then, every loop X involving W_1, W_2, W_3 belongs to tail(P).*

Proof. Follows directly from the definition, since it requires a loop to belong to the *tail*(P). □

4.4.2 Algorithm

In the previous section, a formal definition of loops was provided. Given a set of words L=$\{W_1, W_2, ..., W_k\}$, one can recursively apply definition 4.4.3 on L for adding loops to words $W_1, W_2, ..., W_k$. This process stops when a recursion does not change L. I implemented this as follows:

1. Create a set of all subsequences of each word in L.

2. Starting from the shortest subsequence, create a set of words that contains this subsequence.

3. Apply the loops definition on the set created at step 2.

4. Stop when no loops are added to any word in L.

This process will eventually terminate. This is because the number of words in L is finite and thus the set of all subsequences of words in L created in step 1 is finite. In addition, at each position a word can only have one loop of a given length, i.e., a word does not have two loops of the same length starting at the same position. This suggests that after adding a loop to a word, this loop cannot be added again to the same position in the same word. As a result, eventually all possible loops at all possible positions will be added, which will terminate this process.

Next, I present an algorithm that identifies and adds loops (as defined in 4.4.3) to a Session State Machine. The algorithm assumes that each node knows its level. The start (root) node is at level 0, the start node's children are at level 1, and so on. This is a preliminary version of an algorithm, with optimizations and engineering left for future work. In fact, the only optimization included in this algorithm is that it restricts the possible nodes in a Session State Machine where loops may exist. However, this is the algorithm used for testing the effectiveness of loops definition on real examples and thus is included in this work.

Definitions:

- PQ: a priority queue of nodes ordered by their level. Nodes with lower level have bigger priority. This queue is used for maintaining all the nodes of the tree that have two or more children. These are the only places where a loop might exist. Given a tree, the PQ can be updated by a single traversal of the tree.

- *ancestors*(i): given a node, it returns all its ancestors at distance i from the current node. If it does not exist (e.g. root does not have an ancestor, root's children only have *ancestor(1)*, ...), it returns null. The function *node.ancestors*(0) returns the current node.

- *descendants*(i): given a node, it returns its descendants at i hops away. If it does not exist (e.g. leafs do not have any descendant), it returns null. The function *node.descendants*(0) returns the current node.

- *descendant*(*path*): given a node, it returns its descendant following the path. If the path is not valid, it returns null.

- *path*($node_1, node_2$): returns a path that connects $node_1$ with $node_2$ following child edges, where $node_1$ is an ancestor of $node_2$.

- *merge*($node_1, node_2$): Merges $node_1$ and $node_2$, where $node_1$ is an ancestor of $node_2$, creating a single node. The parents of the new node is the union of the parents of $node_1$ and $node_2$. If $node_1$ is the parent of the $node_2$, $node_1$ is not included in the list of parents of the node derived by merging $node_1$ and $node_2$. Similarly, the set of children of the new node is the union of the children of $node_1$ and $node_2$. If $node_2$ is a child of $node_1$, it is not included in the list of children of the node derived by merging $node_1$ and $node_2$. This is a recursive function, since all equal edges that $node_1$ and $node_2$ have should be followed and recursively merge their children.

Loops Algorithm:

WHILE $PQ \neq empty$

currentNode = first node in PQ;

remove first node from PQ;

FOR i=1 to MaxLengthOfALoop

//All the ancestors at distance i from current node.

ancestors = currentNode.ancestors(i);

//All the descendants at distance i from current node.

descendants = currentNode.descendants(i);

IF($\exists$ [$n_1 \in descendants$ AND $n_2 \in ancestors$

: $path(n_1, currentNode) = path(currentNode, n_2)$]

AND $n_2.getChildren.size() \geq 2$

AND $\exists[n_3 \in n_2.descendants(i) : path(n_2, n_3) = path(currentNode, n_2)]$)

//Path used above.

$path = path(currentNode, n_2)$;

$merge(currentNode, n_2)$;

// This checks if the current node is equivalent with its parent.

// In that case that loop that will be created is X^* instead of X^+

// If current node is equivalent with its parent they should be merged.

IF($currentNode.equivalent(n_1)$)

$merge(n_1, currentNode)$;

END IF

IF(any node before merge had one child and after merge has more that one child)

add node to PQ;

END IF

END IF

END FOR

END WHILE

Merge Algorithm ($node_1$ absorbs $node_2$):

merge($node_1$, $node_2$){
  equal_child_edges = child_edges($node_1$) $\bigcap$ child_edges($node_2$);
  merged_node = create new node;
  merged_node.children = children($node_1$) $\bigcup$ children($node_2$) - $node_2$;
  merged_node.parents = parents($node_1$) $\bigcup$ parents($node_2$) - $node_1$;
  $node_1$ = merged_node;
  FORALL (equal_child_edges)
    //Assuming child.node gives the child node after following an edge
    merge($child.node_1$,$child.node_2$);
  END FORALL
}

4.4.3 Discussion

For testing the effectiveness of the proposed algorithm, a tool that creates a state machine containing loops given a set of words was implemented. Words could be provided either directly or through a state machine. In the latter case, the program utilizes the state machine for creating a set of words. More precisely, starting from the initial state, the algorithm traverses the tree by randomly selecting and traversing one of the outgoing edges of the current state. When a final state is

reached, the path followed is outputted. The outputted string corresponds to a word. By utilizing the second approach (i.e., the input was a state machine), one is able to compare the original with the created state machine, and thus test the effectiveness of the algorithm.

Results suggested that the algorithm was extremely efficient when words were created randomly by utilizing a state machine. Only with a few dozens of words, the algorithm was able to create a state machine identical to the original one. In addition, the algorithm was tested with actual words (i.e., subsequences of a network trace) acquired by monitoring a running DHCP server. In DHCP, a session begins with a DISCOVER message sent by a client, probing a DHCP server to provide an IP address. In a DHCP trace, I observed that a session starts with either a single or a fixed number of DISCOVER messages. More precisely, a client was either able to acquire an IP address with a single DISCOVER message or gives up when it does not receives any response from the server after sending a fixed number of DISCOVER messages. However, based on definition 4.4.3 one cannot consider what described above to be a loop. This suggests that in real examples, more observations will be required for identifying loops. This is because network protocols do not exhibit the same random behavior as the function used by the program for creating random words.

Chapter 5

Validating Models

In previous chapters, I described an approach that creates a state machine called Session State Machine from observations acquired by monitoring the behavior of a network implementation. For validating the generated model, and thus proving the correctness of the implementation, further steps must be taken. The first is to translate the state machine to a formal language, Timed Input Output Automata (TIOA) in our case. Having a TIOA model, the Tempo [57] toolkit can be utilized for translating the TIOA model to either UPPAAL [39] for model checking or TAME [9, 35] for theorem proving. Theorem proving or model checking can then be used for establishing the correctness of the model. In section 5.1, I describe the translator that translates a Session State Machine to a TIOA model. Then, in section 5.2, I explain how the generated TIOA model can be validated against some properties. In addition, I present two different approaches that simplify the process of proving that properties hold on models. The first is an approach proposed by Leonard and Archer in [43]. This approach aims in reducing the complexity of a proof by simplifying the data structures of the model. The second

is an approach that I proposed that automatically creates and provides to the user a set of auxiliary lemmas that can by used for completing a proof.

5.1 From State Machines to TIOA

For defining a Timed I/O Automaton one must define its signature, its states, and its transitions. The signature section is used for defining the actions of the TIOA model. Using the states section, one can define all the state variables of the TIOA model. Finally, the transition section is used for specifying the state transitions when one of the actions defined in the signature is performed. In addition, TIOA allows the definition of complex types. Such types can be defined in packages preceding the automaton definition. Complex types can include new structures, mappings, tuples, and arrays. In what follows, I describe how the translator creates each of these sections of a TIOA model from a Session State Machine.

First, the translator creates the complex types that precede the automaton definition. Those include a special type of queue, the *timed queue*. Each element of the *timed queue* is a pair <time-stamp, message>, i.e., a message with its time-stamp. In addition, complex types are used by the translator for defining messages. More precisely, messages are represented as tuples, where components of the tuple are the regular and symbolic fields of the original message.

The *signature* section of a TIOA model created by the translator contains a list of action signatures. The set of actions is the same for all Timed I/O Automata defined by the translator. It contains one *send* and one *receive* actions. Both actions have a single parameter m of type *Message* that represents the message

sent or received respectively.

The *states* section of a TIOA model generated by the translator has four state variables. The first is a state variable, called *curState*, that keeps track of the current state. Values of this state variable correspond to states of the session state machine. The initial value of this variable is 0, since the initial state of the session state machine is S_0. The second state variable is a *queue* of type *timed queue* and represents the history, i.e., all the messages sent and received and the time that each occurred. After performing a successful transition, the state variable *curState* is updated with the new state to which the transition leads. In addition, the message sent or received is added to the *queue* along with the time that the corresponding event happened. The other two state variables are not related to transitions. Rather, they are used for keeping track of the time. The first one is called *clock*, and it simulates the clock of the system. The second is called *delay*, and is used for delaying the execution of an action so that actions are not executed prematurely.

Timed I/O Automata allow overloading of actions, i.e., actions with the same name, by adding a *where* clause after the definition of an action. In the *where* clause, one can specify the messages that can be handled by the action. For example "*action(m) where m=1*" handles only messages whose value is 1 (assuming the m is of type Int). As mentioned earlier, only two actions are defined in the signature section of the TIOA model, one send and one receive action. However, for creating the *transitions* section of the TIOA model, the translator takes advantage of this feature of TIOA for creating a collection of send and receive actions. Each send and receive action is distinct, because of the different *where* clauses each one has. The translator creates one *send(m)* action of each edge in the session state

machine carrying an output message, and one *receive(m)* action for each edge of the state machine carrying an input message. In both *send(m)* and *receive(m)* actions, the *where* clause contains the label of the edge, i.e., the actual values of the regular fields and the symbolic values of the symbolic fields.

Finally, for allowing transitions only when the model is in a desirable state, all receive actions contain an *IF* statement that checks the current state. State transitions are only allowed if the current state of the model matches the state included in the *IF* statement. Similarly, all send actions have a *precondition* that checks the current state of the model for the same reason.

The Timed I/O Automaton defined above is a template that can be used for creating IOA/TIOA models. This model can be modified based on the property that the tester wants to prove. Examples are presented in section 5.2, where I describe different proofs made on different models.

Appendix A has an example of a TIOA model created by the translator described above.

5.2 Proving Automaton Properties

The final step of the proposed network methodology is to establish that some properties hold on the model. These properties must be extracted from the specification of the implementation and manually added to the model. Having a model and a set of properties, the tester can prove that these properties hold on the model by using either model checking or theorem proving. If the tester can establish that the properties hold on the model, then either testing can stop or more observations must be collected for creating a more complete model, and

thus increase his/her confidence that the implementation is correct.

In the case that the proof fails, i.e., the property does not hold on the generated model, further steps should be taken in order to guarantee that the property does not hold on the IUT as well. This is because the model is an abstraction of the IUT, and as a result, a property that does not hold on the model might hold on the IUT. If model checking is used, a counterexample will be provided to the tester. The tester should then verify that the implementation does not accept the counterexample by applying it to the system. If theorem proving is used, the user should utilize some information extracted from the theorem prover. The first is the queue that keeps track of the history, i.e., all the messages sent and received. The second is the action that the user failed to prove, i.e., after performing the action the property is violated. Using the history, the user can bring the IUT to a state that corresponds to the current state of the model. Then, the action that violated the property must be executed in the IUT. If the IUT also violates the property, a bug in the implementation is found. In different cases, only the model violates the property and the user must reconsider the abstractions used for building the model.

This work addresses only theorem proving[1]. More precisely, it tries to address the difficulty of making proofs using theorem proving. Prototype Verification System (PVS) [51] is the theorem proving system used in this work. Having a TIOA model of the implementation along with a set of properties that must hold, TIOA to PVS converter [44] can be utilized for translating the TIOA to a PVS model. The converter was implemented by TDS within the theory group at MIT. For

[1]Model checking is automatic and thus the difficulty level of making proofs is considerably lower than that of theorem proving.

making proofs, TAME [8, 7] is utilized. TAME is an interface to PVS designed for proving properties of automata. It is equipped with strategies specifically designed for reasoning about automata. The use of these strategies can be used as a means for simplifying proofs. Despite the use of TAME and the leverage it provides to proofs, initial attempts for proving some properties revealed some challenges. Proofs for even simple properties on small models were very hard and time consuming. Equally time consuming was the identification of auxiliary lemmas necessary for completing a proof, since this required both deep understanding of the model and deep knowledge of PVS. These results suggested that approaches that simplify the process of proving that properties hold on models must be investigated and adopted.

For simplifying proofs using theorem provers, I collaborated with Myla Archer and Elizabeth Leonard from the Center for High Assurance Computer Systems (CHACS) of the Naval Research Laboratory (NRL). The approach they suggested was to utilize the reference variable approach proposed by Leonard and Archer in [43]. For testing the effectiveness of this approach on models that represent network implementations, this method was applied to three different cases. Results suggested that this approach can effectively reduce the complexity of proofs. The case studies used along with all the results acquired are presented in section 5.2.1. In addition, in section 5.2.1.4, I discuss the results along with the knowledge gained for these case studies.

In addition to the *reference variable* approach, I proposed another approach that assists the generation of auxiliary lemmas. This latter approach utilizes the fact that both the models and the properties in the context of this work belong to a specific class. Restricting the class of programs investigated to network imple-

mentations, and also creating models using a specific template, i.e., the template defined in the previous section, one can create stylized models, all sharing similar structure. A class of properties that must hold on network implementations can also be defined. This is because properties of network implementations are defined in terms of permissible or non-permissible subsequences of messages. Utilizing the assumption that models and properties belong to specific classes, one can make predictions about their structure, which facilitates the creation of more effective approaches. This approach along with three case studies used for testing the effectiveness of the approach are presented in section 5.2.2.

5.2.1 Reference Variable Approach

The goal of the reference variable approach is to simplify the process of proving properties by referring directly to information vital for the proofs and thus avoiding references to fields buried in complicated data structures. This is achieved by providing some special variables called *shadow variables* that independently maintain some of the same information contained in other, more complex data structures in the model. After defining them, many properties can be specified in terms of the simpler shadow variables, resulting in simpler proofs.

In this section, I focus on shadow variables and how they can simplify proofs. The models of three different protocols, DHCP, TCP, and STP are used. All three models were created from running systems. For testing the effectiveness of this approach, a comparison of the complexities of proving equivalent properties on the original model and a model where shadow variables were introduced is made. The measurements include the following:

- run time: the actual time that PVS required for completing a proof.
- depth of the proof tree: an estimation of the complexity of subgoals considering that more complicated subgoals require more steps, which results in deeper proof trees.
- subgoals of the proof tree: an estimation of the complexity of the property and the model, given that more complicated properties or models require more subgoals.
- leafs of the proof tree: the difference between the number of subgoals and the number of leafs gives an estimation of the complexity of subgoals. This is because complicated subgoals are usually split into simpler sub-subgoals.
- auxiliary lemmas: this is another measurement of the complexity of the model and the property. This is because more auxiliary lemmas are required when facts about the model are not obvious during the proof, i.e., complicate models.
- man hours: the time in hours a proof required.

In what follows I present my findings from the use of shadow variables in the three models. Before presenting the results, I give a brief description of each protocol's functionality, the property that the model must satisfy and the Session State Machine used. The term *original* model refers to a PVS model equivalent to the Session State Machine, and *original* property to the property that must hold on the *original* model. Similarly, the term *shadow* model refers to a model isomorphic to the *original* model, where shadow variables are added. In addition,

the term *shadow* property refers to an equivalent to the *original* property, which is the property that must hold on the *shadow* model.

5.2.1.1 Dynamic Host Configuration Protocol (DHCP)

DHCP [3] is a protocol that automatically assigns IP addresses to clients. When a client joins an IP network, it requests a valid IP address from a local DHCP server. Each DHCP server handles a pool of IP addresses and, if it has at least one available, it offers that to any client that requests an IP address. Clients can use an IP address for a specific amount of time, known as *lease time*. If the client wants to use the IP address for a longer period, it can extend the lease time by renewing the IP address. The most important property of DHCP is that no two clients should have the same IP address for overlapping time. Having this in mind, a model was build, which captures the component of a DHCP server that is responsible for tracking the availability of a particular IP address. In other words, it shows how the server assigns the same IP address to different clients over time. A state machine that corresponds to this model is presented in figure 5.1.

The assignment of an IP address to a client by the server is confirmed at the moment the client receives an acknowledgement message. The server can reuse the same IP address (i.e., assign it to a different client) only after the expiration of its *lease time*, i.e., the period that the client can use the IP address.

For proving the correctness of the above property, it is enough to show that the model can never generate two acknowledgment messages having as recipients two different clients in time less than the *lease time*[2]. In other words, if two

[2]Have in mind that the model generated corresponds to a single IP address. This is why the property does not check the assigned IP address.

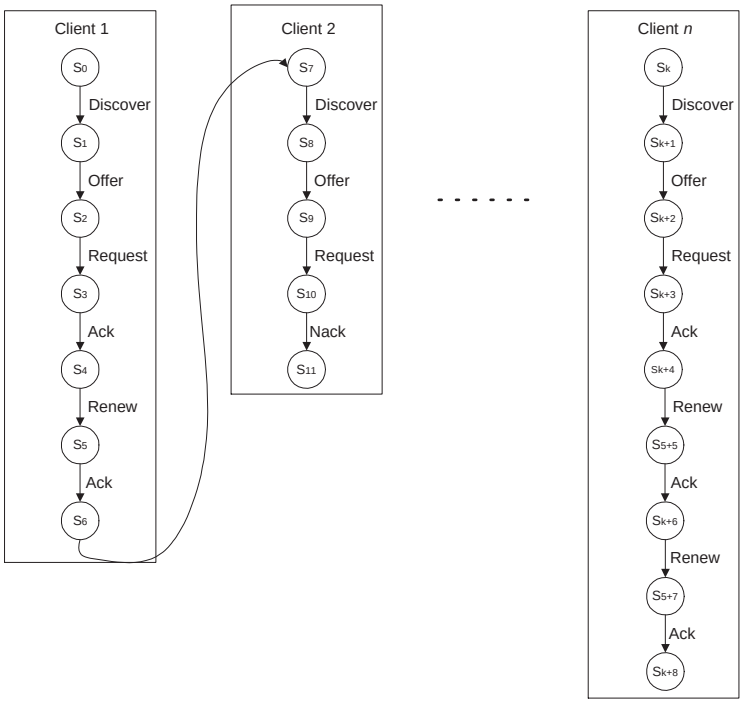

Figure 5.1: DHCP Model - Assignment of the same IP address to different clients.

acknowledgement messages are sent to two different clients (the hardware address, i.e., the value carried by the *chaddr* field, of the two messages should not be equal) then the time stamp on the messages should be apart at least *lease time*. One can state this property in TAME as follows:

$$chaddr(message(second\_lastP(queue(s), msgOfTypeAck?))) \neq$$
$$chaddr(message(lastP(queue(s), msgOfTypeAck?)))$$
$$\Rightarrow$$
$$timestamp(second\_lastP(queue(s), msgOfTypeAck?)) + lease\_time <$$
$$timestamp(lastP(queue(s), msgOfTypeAck?))$$

where:

- $queue(s)$: is the queue of timed messages, i.e., $< timestamp, message >$, at state s.

- $msgOfTypeAck?$: is a predicate that returns true if the type of the message is *acknowledgment*.

- $lastP$: is a function that takes two arguments, a queue and a predicate, and returns the most recent element in the queue that satisfies the predicate. As defined above, it will return the most recent acknowledgment sent.

- $second\_lastP$: is a function similar to one defined above, but it returns the second to the last element that satisfies the predicate. As defined above, it will return the second most recent acknowledgment sent.

- $message$: returns the message part of a timed message.

- $timestamp$: returns the time stamp part of a timed message.

- $lease\_time$: is the period that the IP address can be used by the client.

For testing the effectiveness of shadow variables, an additional model whose behavior was the same (i.e., same transitions) to the original one was created. Two shadow variables were introduced to this model, last_ack and second_last_ack. These two shadow variables were used for keeping track of the last and the second to last acknowledgment messages sent by the server. Additional changes were made to the model for updating the values of these shadow variables. These changes were trivial; whenever the server sent an acknowledgement message, the value of the last_ack was copied to second_last_ack, and the new acknowledgement message was copied to the last_ack.

Using the two shadow variables, the *original* DHCP property was reduced to the following *shadow* property:

$$chaddr(message(last\_ack(s))) \neq chaddr(message(second\_last\_ack(s)))$$

$$\Rightarrow$$

$$timestamp(last\_ack(s)) + lease\_time < timestamp(second\_last\_ack(s))$$

The findings from this example (table 5.1) clearly illustrate the effectiveness of this approach on the DHCP model.

	original	*shadow*
run time	42.70 secs.	5.02 secs.
proof tree max. depth	25	5
proof tree subgoals	18	5
proof tree leafs	29	5
auxiliary lemmas	11	6
man hours	10.5	1.5

Table 5.1: Complexities of the proofs of the *original* and *shadow* properties of DHCP.

5.2.1.2 Transmission Control Protocol (TCP)

TCP [33] is one of the most heavily used protocols in networks. This is mainly because it is the only protocol that provides applications with reliable and in order message delivery. Here, I focus on the retransmission behavior of TCP, which is responsible for the reliable delivery of messages. More precisely, I proved the TCP properties that: (a) every message sent eventually gets acknowledged; and, (b) no retransmissions of an acknowledged message occur. Traces from a running network were collected and utilized for creating a model that captures this behavior. Messages sent with the same sequence number (i.e, initial transitions and

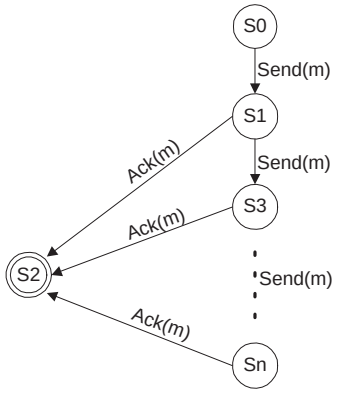

Figure 5.2: TCP retransmission Model.

all the retransmissions of a message) were grouped with the first acknowledgment received that acknowledges this message (i.e., a message with acknowledgment number greater that the sequence number of the message sent). The model generated is similar as the one presented in figure 5.2, which essentially shows that the sender keeps sending the same message until an acknowledgment is received.

The proof of the second property, i.e., no retransmissions of a message that has been acknowledged, is trivial. This invariant can be proved by checking if the last message in the queue at the final state is an acknowledgment message. Because of the simplicity of this property (only checks a single message), the proof was trivial on both the *original* and the *shadow* models. For this reason, the results of proving this property are not included here. For proving the first property, one must prove that the acknowledgement message received acknowledges the correct message, i.e., the message sent. This is true if the acknowledgement number carried by the acknowledgment message is greater than the sequence number of the message sent. This property can be defined in TAME as follows:

$$lastP(queue(s), AckMsg?) \mathrel{/}= Bottom \text{ AND}$$
$$lastP(queue(s), SeqMsg?) \mathrel{/}= Bottom$$

$$\Rightarrow$$

$$seq(message(lastP(queue(s), SeqMsg?))) <$$
$$ack(message(lastP(queue(s), AckMsg?)))$$

where:

- $lastP$: a function that takes two arguments, a queue and a predicate, and returns the most recent element in the queue that satisfies the predicate. As defined above, it will return the most recent acknowledgment sent.
- *Bottom*: Bottom is used in the same way that *null* is used in programming languages. Here is used for checking if $lastP$ function returns an element of the queue.
- $queue(s)$: the queue at state s.
- $AckMsg?$: a predicate that returns true if the message is an *acknowledgment* message.
- $SeqMsg?$: a predicate that returns true if message is a *send* message.
- *seq* : the field that carries the sequence number of a *send* message.
- *ack* : the field that carries the acknowledgment number of an *acknowledgment* message.

To create the *shadow* model, two shadow variables, *sentMsg* and *receivedMsg*, were added to the original model. The former was used for holding the message sent and the latter for holding the message received. Using these two shadow variables, the *original* property was reduced to the following *shadow* property:

$$sentMsg \mathrel{/}= Bottom \text{ AND } receivedMsg \mathrel{/}= Bottom$$

$$\Rightarrow$$

$$seq(message(sentMsg)) < ack(message(receivedMsg))$$

Table 5.2 summarizes the findings from this example. The results show that the shadow variables reduced the complexity of proving the desired property on a TCP model.

	original	*shadow*
run time	29.80 secs.	17.75 secs.
proof tree max. depth	11	4
proof tree subgoals	44	43
proof tree leafs	44	43
auxiliary lemmas	5	2
man hours	5.5	1

Table 5.2: Complexities of the *original* and *shadow* properties of TCP.

5.2.1.3 Spanning Tree Protocol (STP)

STP [31] is a protocol that eliminates cycles from a bridge network by blocking the traffic on some links. The links that remain active after running the STP protocol on a network form a spanning tree. The work presented here concentrates on the leader election part of the protocol. This part requires all participating bridges to elect a unique leader, known as *root bridge* (essentially the root of the spanning tree). To elect a leader, periodically (i.e., in rounds), each bridge sends a message to all its neighbors advertising the id of the bridge it believes it is the root bridge. As a result, at each round a bridge receives messages from all of its neighbor bridges. At the end of the round, a bridge has to decide on the root bridge based

on the messages received during the round, and send out a new message with updated information about the new root bridge calculated.

Figure 5.3 represents a Session State Machine generated from a STP implementation. Each path from state S0 to S1 represents a round, where an incoming edge at state S1 always represents the message sent by the bridge, and all the other messages represent messages received by the bridge during a round. The first argument of the send and receive actions is a number that represents the id of the bridge that generated the message, and the second is a number representing its computed root bridge.

Special attention must be paid to the path $S0 \rightarrow S4 \rightarrow S5 \rightarrow S1$. This path represents an execution observed in a trace generated by a real implementation, which is not clearly defined in the specification. In this execution, during the same round, a bridge received two different messages coming from the bridge with id 3. In cases when a bridge receives more than one messages from the same source, only the last message received should be taken into consideration when electing the root bridge.

The desired property to be proved is that each bridge always advertises the "correct" (defined below) bridge as the root bridge. In order to prove that, two fields of the STP message were utilized, the *root_id* field, which is the field used by STP for propagating its computed root bridge and the *bridge_id*, which carries the id of the bridge that generated the message. The correctness of the advertised root can be checked using the following expression:

$$next\_root\_id = min(min\_root(last\_message\_per\_bridge), bridge\_id)$$

where:

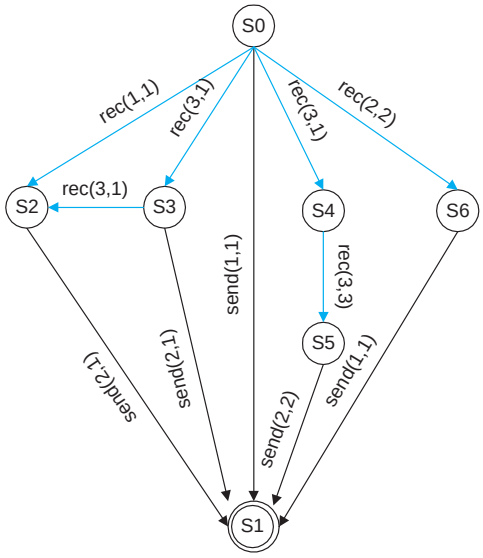

Figure 5.3: STP Model.

- *last_message_per_bridge* : is a list that contains the last message received from each neighbor bridge during a single round.

- *min_root* : returns the message in the last_message_per_bridge list that carries the smallest root_id.

- *min* : returns the minimum between the two arguments that it takes as parameters.

The above property requires an operation that returns the message among the last messages received by each neighbor at each round that carries the smallest root id. However, by using only the state variable that contains this information (i.e., the queue that holds all the messages sent and received at a single round), it is impossible to perform the operation $min\_root(last\_message\_per\_bridge)$. This is because the queue may contain multiple messages received from one or more neighbors. As a result, an additional state variable must be defined, which will

maintain the last message received from each neighbor. A *min* function on this state variable can then be used for extracting the message that carries the smallest root id. The term *augmented* model is used to refer to a model augmented with this state variable. An augmented model derives from the *original* model by adding the minimum information that allows the user to define the property that must hold on the STP model described above.

For creating a model that utilized shadow variables, the *original* model was augmented by defining one state variable for each neighbor bridge, $LastRcvdMsg-Neighbor_1$, ..., $LastRcvdMsgNeighbor_n$. These state variables were responsible for maintaining the last message received by each neighbor. In addition, a state variable that maintained the message that carried the minimum root id, $cur\_min = min(LastRcvdMsgNeighbor_i)$, where $1 \leq i \leq n$, was defined. This simplified the process of finding the message that carried the minimum root id among the last messages received. Also, the property to be proved was reduced to:

$$next\_root\_id = min(cur\_min, bridge\_id)$$

Table 5.3 summarizes the findings acquired from the proofs of the two equivalent models.

	augmented	*shadow*
run time	7.36 secs.	4.32 secs.
proof tree max. depth	7	4
proof tree subgoals	12	12
proof tree leafs	12	12
auxiliary lemmas	8	4

Table 5.3: Complexities of the *augmented* and *shadow* properties of STP.

Clearly, the reduction in complexity was not as dramatic as in the previous two examples. The reasons for this is discussed in the following subsection.

5.2.1.4 Discussion

The use of the reference variable approach on the DHCP and TCP models clearly illustrated that it can simplify every aspect of a proof and also that the overhead for adding shadow variables to the models is negligible. On the contrary, the use of the reference variable approach on the STP model illustrated some different aspects of its use. First, the effectiveness of the reference variable approach is influenced by the model and property that must hold on the model. The reference variable approach was not as effective in the STP example, because the desired property was not defined in terms of the *queue*. However, it is expected that the huge majority of network properties will be defined using the *queue*. In addition, this example illustrated that there are properties where shadow variables must be defined based on the model and not the property. In the STP example, one shadow variable for each neighbor bridge was defined. As a result, models of the same protocol but with different number of neighbor bridges will require different shadow variables.

Another observation that must be reported is the number of subgoals and leafs of the proof trees of the *original* and *shadow* properties. The only model where the number of subgoal and leafs was reduced is the DHCP model (Table 5.1). This is because some of the subgoals of the *shadow* property were trivially true and as a result, TAME discarded those subgoals. However, it has to be made clear that the *reference variable* approach aims at reducing the complexity of proving subgoals and not the number of subgoals.

5.2.2 Automatic Creation of Auxiliary Lemmas

In this section, I propose an algorithm that automatically creates auxiliary lemmas that can be utilized either directly by the user or by a strategy. For creating them, the algorithm takes advantage of the fact that models and properties are stylized and thus have similar structure. More precisely, it utilizes two state variables of the model, the timed message queue, *queue*, and the integer, *curState*. In addition, it assumes that properties are defined in terms of the queue, i.e., sequences of messages that must or must not appear in the queue.

The algorithm is split into three phases. The first phase creates auxiliary lemmas related to the initial values of the state variables. The goal of the second phase is to create auxiliary lemmas that set any of the *antecedent* formulas to false or any of the *consequent* formulas to true. Finally, the last phase creates auxiliary lemmas stating properties of the *queue*. Next, I describe each phase in detail and provide examples illustrating its use.

The first step of the algorithm creates auxiliary lemmas related to the initial values of the state variables of the model. This can be achieved by parsing the file that holds the model definition and in particular, the action that initializes the state variables. For each state variable two auxiliary lemmas will be created. The first will formally define the initial value of the state variable. The second auxiliary lemma will be the *converse* of the first one, where $converse(a \Rightarrow b) = \neg a \Rightarrow \neg b$. For example, assume that the initial value of the state variable *curState* is 0 and that the *queue* is empty, denoted by the predicate *mtQ?*. Using this information, one can create the following two auxiliary lemmas (mind that elements of the *queue* are never removed):

1. IF $curState(s) = 0 \Rightarrow mtQ?(queue(s)) = true$;

2. converse(IF $curState(s) = 0 \Rightarrow mtQ?(queue(s)) = true$) = IF $curState(s) > 0 \Rightarrow mtQ?(queue(s)) = false$;

The second and third phases of this algorithm require some additional input. The input will be a set of predicates that can be used for filtering interesting from non-interesting messages that appear in the *queue*. For example, in DHCP, such predicate will distinguish acknowledgment messages from all the other messages in the queue and it can be defined in TAME as "$msgOfTypeAck?(tm : timed\_message[Message]) : bool = msgType(message(tm)) = 5;$", i.e., a message is an acknowledgement message, if the value of the msgType field is 5. The only assumptions made here is that all the predicates that define an "interesting" message start with the keyword "*msgOfType*".

It is also important to mention that these predicates are not defined specifically for this algorithm. Instead, they are defined by the user to allow him/her to define properties. The only assumption is that all the predicates of this form are defined in a single file that the algorithm can access and parse.

In more detail, this phase of the algorithm creates sets of states along with properties that hold on these states by setting any of the *antecedent* formulas to false and any of the *consequent* formulas to true. I explain this phase in more detail through the use of an example. Assume that one tries to prove the DHCP property that no two clients have the same IP address at the same time (also used in section 5.2.1.1) on the DHCP model presented in figure 5.4:

$$chaddr(message(second\_lastP(queue(s), msgOfTypeAck?))) \neq$$
$$chaddr(message(down(lastP(queue(s), msgOfTypeAck?))))$$

$$\Rightarrow$$

$$deadline(second\_lastP(queue(s), msgOfTypeAck?)) + lease\_time <$$

$$deadline(down(lastP(queue(s), msgOfTypeAck?)))$$

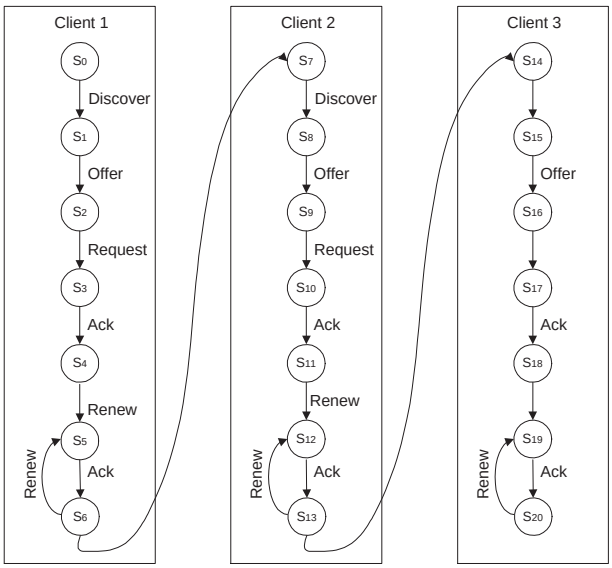

Figure 5.4: DHCP Model - Assignment of the same IP address to three different clients.

Based on the model, three sets of states will be created that set the *antecedent* formulas to false, one for each client. These sets are $Set_1 = \{S_0, S_1, ..., S_{10}\}$, Set_2=$\{S_{12}, S_{13}, ..., S_{17}\}$, and Set_3=$\{S_{19}, S_{20}\}$. The first set contains the states where the last two acknowledgment messages in the *queue* (if exist) are sent to the first client, the second set contains the states where the last two acknowledgment messages in the queue are sent to the second client and so forth.

Similarly, sets of states where properties that set the *consequent* formulas to true will be created. In the above example, two sets will be created $Set_3 = \{S_{11}\}$ and $Set_3\{S_{18}\}$. The property that holds in these states that set the *consequent* formulas to true is that the last two acknowledgment messages in the *queue* are

apart at least *lease time* (*lease time* is defined in section 5.2.1.1). Based on the above, the following auxiliary lemmas will be created:

1. Properties that set any of the *antecedent* formulas to false:

   (a) $IF(state \in Set_1) \Rightarrow$

   $chaddr(message(second\_lastP(queue(s), msgOfTypeAck?))) = client_1$

   AND

   $chaddr(message(down(lastP(queue(s), msgOfTypeAck?)))) = client_1$.

   (b) $IF(state \in Set_2) \Rightarrow$

   $chaddr(message(second\_lastP(queue(s), msgOfTypeAck?))) = client_2$

   AND

   $chaddr(message(down(lastP(queue(s), msgOfTypeAck?)))) = client_2$.

   (c) $IF(state \in Set_3) \Rightarrow$

   $chaddr(message(second\_lastP(queue(s), msgOfTypeAck?))) = client_3$

   AND

   $chaddr(message(down(lastP(queue(s), msgOfTypeAck?)))) = client_3$.

2. Properties that set any of the *consequent* formulas to true:

   (a) $IF(state \in Set_4) \Rightarrow$

   $timestamp(second\_lastP(queue(s), msgOfTypeAck?)) + lease\_time <$

   $timestamp(down(lastP(queue(s), msgOfTypeAck?)))$

   (b) $IF(state \in Set_5) \Rightarrow$

   $timestamp(second\_lastP(queue(s), msgOfTypeAck?)) + lease\_time <$

   $timestamp(down(lastP(queue(s), msgOfTypeAck?)))$

The final phase of this algorithm creates auxiliary lemmas related to the existence or not of important (defined by the predicates msgOfTypeX?) messages in the queue. As in the first phase, this phase creates an auxiliary lemma that states that no important message exists in the queue at a given state, and also its converse (as defined above). Using the state machine defined in figure 5.4 and the predicate *msgOfTypeAck?*, the following two auxiliary lemmas are created:

1. IF $curState(s) <= 3 \Rightarrow lastP(queue(s), msgOfTypeAck?) = Bottom$;
2. IF $curState(s) > 3 \Rightarrow lastP(queue(s), msgOfTypeAck?) /= Bottom$;

To test its effectiveness, this algorithm was applied to the *original* model of the DHCP and TCP protocols and the *augmented* model of the STP protocol defined in section 5.2.1. Table 5.4 presents the findings from this example. The column *Required* defines the actual number of auxiliary lemmas required, and the column *Created* defines how many of the required lemmas were created by this algorithm.

Protocol	*Required*	*Created*
DHCP	11	8
TCP	5	4
STP	8	2

Table 5.4: Results from the use of the algorithm that creates auxiliary lemmas.

5.2.2.1 Discussion

The results showed that this approach worked very well for models where a property was defined using the state variable *queue*, i.e., DHCP and TCP protocols.

However, for STP the only two auxiliary lemmas utilized for proving the property were the ones generated by the first phase of the algorithm, i.e., auxiliary lemmas related to the initial values of the state variables. This is because the *augmented* property for STP was not defined in terms of the queue. As a result, all the auxiliary lemmas created by phases two and three were useless, given that these phases created invariants related to the status of the *queue* at different states.

Chapter 6

Using Database Queries for Extracting Words

Creating a Session State Machine from sequences of messages (i.e., observed network behaviors that are called *traces*) requires breaking each original sequence into a collection of subsequences. In section 4.1, I defined such subsequences as *sessions*. Each session is regarded as a word of the language accepted by the generated state machine. In this chapter, I present an approach that is capable of creating more general sessions from a sequence of messages. This approach utilizes existing database technology and *Parameterized Queries*, an extended form of a relational algebra query defined for this purpose. In addition, a class of *Parameterized Queries* that can be computed efficiently is defined.

6.1 Definitions

Before formally defining *Parameterized Query*, some necessary terminology must be defined. A relation R can be thought of as a table whose columns are called *Attributes* and rows *Tuples*. Denote $R(A_1, \ldots, A_n)$ to be a relation R over the set of $attributes(R) = \{A_1, \ldots, A_n\}$, having domains $dom(A_1), \ldots, dom(A_n)$. A query retrieves information from R and is defined as $\pi_{A_1,\ldots,A_n}\sigma_\varphi(R)$, where:

1. $\pi_{A_1,\ldots,A_n}$ is called projection operation and defines the attributes of R returned by a query.

2. σ_φ is called selection operation and φ is a propositional formula that is used for selecting the tuples of R returned by a query. It is formed from a set of symbols taken from the set $\{attributes(R) \bigcup dom(A_1) \bigcup \ldots \bigcup dom(A_n)\}$ and the operations AND, OR and NOT.

As mentioned earlier, the motivation for this work was to create arbitrary slices of a relation R. However, a database query returns only one result; a slicing requires many results. For this purpose, *Parameterized Query* is defined, which is an extended form of a relation algebra query capable of slicing a relation R in arbitrary ways.

Definition 6.1.1. A *Parameterized Query* is a query that can include symbols from the set $\{\$i \mid i \in \mathbb{N}\}$ in place of domain values.

Based on the above definition, a *Parameterized Query,* is a new type of query including symbols that can be instantiated with many different values, e.g. $\pi_a \bullet \sigma_{a<\$1}(R)$. Since the formula $a<\$1$ is only defined on values in $dom(a)$, one can restrict the values to be considered to $dom(a)$.

In general, one can think of a *Parameterized Query* as the set of ordinary queries formed by assigning different values to its parameters $p_1, \ldots, p_n$. Each different assignment is called *instantiation*, and each *instantiation* results in a different ordinary query, which is called *instantiated parameterized query*. An *instantiated parameterized query* returns a set of tuples, hereafter defined as *slices* (a *slice* is similar to a *session* as defined earlier). A Parameterized Query creates a set of *slices*, one for each different instantiation that returns at least one tuple; an instantiation that returns no tuples is not considered a slice.

For running the Parameterized Query (PQ) on a relation R, each parameter p that appears in PQ must be instantiated with a value from an appropriate domain. However, for defining the domain of each parameter p, one must define the set of all attributes that appear in PQ such that PQ has a term of the form (a *op* p), where a is an attribute, *op* is an operator and p is a parameter. Formally, $Attr(p) = \{A \mid$there is a term of the form (a *op* p) in σ $\}$.

Definition 6.1.2. The domain of a parameter p is the set $Values(p) = \{T(p)\}$, where $T(p) = \bigcup_{A \in Attr(p)} \pi_A(R)$, i.e., all the distinct values appearing in the database of attributes associated with p. For a parameterized query to represent a non-empty slicing, it must have identical domains in *Attr(p)*.

Example 6.1.3. This is an example of the use of the Parameterized Query $PQ = \pi_{a,b} \bullet \sigma_{a<\$1}(R)$. The first step is to create the domain of parameter *$1*, $Values(\$1) = \{1, 2, 3\}$ (See Table 6.1). The first relation in the figure, R, represents the original relation where PQ was applied to. All the other relations represent the results that PQ returns when instantiating *$1* with values that belong to *Values($1)*.

□

R	
a	b
1	3
2	2
3	4

$\sigma_a<1(R)$ ($\$1=1$)	
a	b

$\sigma_a<2(R)$ ($\$1=2$)	
a	b
1	3

$\sigma_a<3(R)$ ($\$1=3$)	
a	b
1	3
2	2

Table 6.1: Example of calculating a *Parameterized Query*.

6.2 Creating Slices using Parameterized Queries

Conceptually, the easiest way to create slices using a Parameterized Query is to create all possible combinations of values that can be used for instantiating its parameters, i.e., $Values(\$1) \times Values(\$2) \times \ldots \times Values(\$n)$. Each different combination creates a different instantiated parameterized query. However, this approach has three major drawbacks: its complexity, the generation of identical slices, and the generation of empty slices.

This section addresses these issues by answering the question, "How can one determine the minimum number of queries required to retrieve all the slices for a given parameterized query?". On the basis of the answer to this question, an algorithm that can be used for evaluating a parameterized query is proposed. However, before describing the algorithm, some assumptions that are considered necessary for overcoming a problematic case are discussed.

Without loss of generality, assume that the propositional formula of a query is given in a Disjunctive Normal Form (DNF). In this way, one can simplify the generation of slices by breaking the process into two steps:

1. For each conjunctive clause c of the disjunctive normal form, retrieve the tuples returned by each different instantiation of each parameter in c. This

process will create a set of sub-slices, which is a subset of the set of messages in the final slice.

2. Use the set of sub-slices, created in step 1, to create the final slice by calculating the union of the tuples of each sub-slice.

I argue that for propositional formulas that are in DNF, one can easily find all "useful" instantiations, i.e., instantiations that return at least one tuple, of parameter terms whose operation is '='. It is enough to observe that parameter terms of that type return some tuples only if an instantiation matches a tuple of R. To clarify this, consider the following example:

Example 6.2.1. Assume the parameterized query $\pi_{a,b,c} \bullet \sigma_{a=\$1 \wedge b=\$2}(R)$ and the relation R:

	a	b	c
1	3	4	2
2	4	5	2
3	4	5	5

The domains of the two parameters \$1 and \$2 are respectively $Values(\$1) = \{3, 4\}$ and $Values(\$2) = \{4, 5\}$. However, creating one instantiated parameterized query for each combination of values of these two sets is wasteful, because most of them return no results. Instead, for instantiating \$1 and \$2, it is enough to use the set returned by the operation $\pi_{a,b}(R)$, which is equal to the minimum set that can be used for instantiating PQ so that it returns at least one tuple. □

Based on the above results, one can conclude that the minimum set of values for instantiating parameter terms whose operator is '=' can be extracted with a

single query. Thus, in what follows, I focus more on instantiations whose operator is not '='.

I also assume that in each clause, if two parameter terms use the same parameter and the operation is an inequality, then the inequality goes in the same direction. For example, the selection operation $\sigma_{a<\$1 \wedge b>\$1}$ violates the assumption, because the \$1 appears in both parameter terms, but the inequality goes in different directions. On the contrary, $\sigma_{a<\$1 \wedge b=\$1}$ is legal, because the direction of inequality is the same. Without this assumption, empty and non-empty instantiations can be alternated (see example 6.2.2). However, in order to improve the performance of the proposed algorithms, the assumption must be made that when an empty instantiation is found, then no additional instantiation will be found that returns non-empty slices when continuing along the same axis. Example 6.2.2 illustrates a scenario where empty and non-empty instantiations alternate.

Example 6.2.2. Assume the Parameterized Query $\pi_{a,b,c} \bullet \sigma_{a \geq \$1 \wedge b \geq \$2 \wedge c \leq \$1}(R)$ and the relation R:

	a	b	c
1	3	4	2
2	7	6	5
3	4	5	2
4	4	5	6

The domain of the two parameters are $Values(\$1) = \{2, 3, 4, 5, 6, 7\}$ and $Values(\$2) = \{4, 5, 6\}$. The following table presents all possible combinations of the two sets, i.e., $Values(\$1) \times Values(\$2)$, and each cell captures the tuples returned by each one of them.

6	ø	ø	ø	ø	2	ø
5	ø	3	ø	ø	2	ø
4	ø	3	ø	ø	2	ø
$2↑ $1→	2	3	4	5	6	7

Table 6.2: Mapping Table where duplicate cells cannot be removed.

In table 6.2, the instantiations that return non-empty slices are not continuous, i.e., there are empty instantiations of the parameterized query between instantiations that return at least one tuple ([3, 4] returns tuple 3, [4, 4] and [5, 4] return no tuples, and [6, 4] returns tuple 2). □

6.3 Mapping Tables

In this section, the notion of *Mapping Tables* is defined, a notion used for efficiently calculating the results of a parameterized query. A *Mapping Table* is an n-dimensional table, where n is the number of the parameters that appear in a Parameterized Query. The labels on axis i are taken from the domain of the parameter $\$i$, i.e., $Values(\$i)$. If the operation on $\$i$ is $>$ or $=$, then the values to be used for labeling the axes will be placed in ascending order; otherwise, they will appear in descending order. Each cell of the *Mapping Table* corresponds to the set of tuples returned by the Parameterized Query, instantiated with the labels of the axes that correspond to that cell. Conceptually, a completely calculated *Mapping Table* represents the results of executing a Parameterized Query. Mapping tables are closely related to the relational aggregation operation *data cube*, introduced by Gray et al. in [24]. Actually, the approach proposed here is a generalization of the data cube operation. Data cube operation utilizes a table, where each cell

contains an aggregate. Using mapping tables, each cell of a table contains a set of tuples from which different aggregates can be computed.

To define a *Mapping Table*, the domain of each parameter must be found. This requires one query for each term that contains a parameter.

Example 6.3.1. Assuming the parameterized query $PQ = \pi_{a,b,c} \bullet \sigma_{a \geq \$1 \wedge b \leq \$2}(R)$ and the relation R:

	a	b	c
1	3	4	2
2	4	5	2
3	4	6	5

Given that PQ has two parameters ($1 and $2), this requires a 2-dimensional Mapping Table, where *Values*($1) = {3, 4} and *Values*($2) = {4, 5, 6}:

4	1	ø
5	1,2	2
6	1,2,3	2,3
$2↑ $1→	3	4

Based on the above Mapping Table, PQ will return 5 different slices for instantiations [3, 6], [3, 5], [3, 4], [4, 6], and [4, 5]. □

The use of mapping tables is proposed because it facilitates efficient calculation of Parameterized Queries. First, in any row going left to right, once an empty cell appears, subsequent cells will always be empty. Second, after calculating the edge cells, no additional queries on R are necessary for creating the internal cells. Finally, cells above and to the right of a given cell are subsets of it. This is because

the values of an internal cell are the intersection of its neighbor cells. In the above example, the value of cell [4, 5] is the intersection of the values of cells [3, 5] and [4, 6].

6.4 Creating a Mapping Table

In this section, I present an algorithm that creates a mapping table and outputs the slices derived from it. It assumes that each cell of the table maintains a set of tuples, i.e., each cell represents a different slice.

6.4.1 Definitions

Central in the algorithm is the notion of *level*, which is defined as the distance of an axis has from the edge axes. Edge axes have level 1; axes adjacent to edge axes have level 2, and so on. Also the term level is used to refer to a set of cells, i.e., cells at level *i*. Also, it is important to observe that all axes that belong to the same level share a common cell, the *base cell* (figure 6.1).

For reducing the size of the memory that the algorithm uses, the algorithm does not maintain into the memory the complete mapping table. Instead, it maintains the cells of two adjacent levels. In addition, it uses vectors for maintaining the axes of each level. This approach allows axes of different length in different levels. As a result, empty cells are removed from the axes and are not maintained in the memory.

Levels whose cells are maintained in the memory are called *active levels* and the axes of the active level, *active axes*. As mentioned earlier, there are at most two active levels at any given time, namely *current_level* and *next_level*. Assuming that

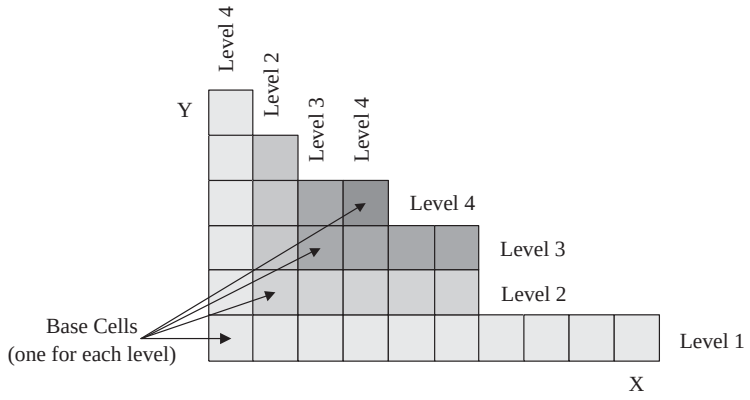

Figure 6.1: Levels and Common Cells.

the dimension of the mapping table is n, then for each level n vectors are required. In figure 6.1, two vectors are required for each level. One vector maintains the values along the X axis and the other along the Y axis. After creating the cells of the next level and saving the cells (i.e., slices) of the current level to a file, then the next level is set to be the current level (this also releases the memory occupied by the cells of the current level).

Axes that have a single element can be eliminated from the set of axes. This is because all axes of the same level share a common cell, the *base cell* (see figure 6.1). As a result if the size of an axis is one, one can conclude that the only cell the axis has is the base cell and thus it can be removed from the set of axes. This also reduces the dimension of the mapping table by one, since one of the axes is eliminated.

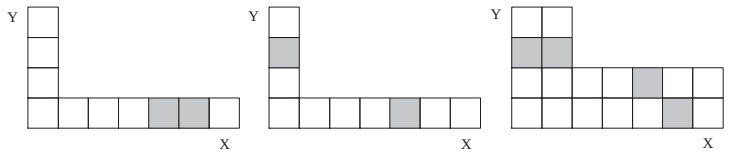

Figure 6.2: Duplicate cells of type 1, 2 and 3 respectively.

6.4.2 Duplicate Cells

Cells that carry the same set of tuples might exist in a mapping table. However, the complexity of the algorithm can be reduced by eliminating, when it is possible, duplicate slices. The term *duplicate cells* is used to refer to cells that carry the same set of tuples. In cases where a mapping table contains duplicate cells, only one of them should be saved into the slices file. Otherwise, duplicated slices will appear in the output file. Duplicate cells are grouped into three categories based on the position they appear in the Mapping Table:

1. Duplicate cells that appear along the same axis.

2. Duplicate cells that appear along different axes of the same level.

3. Duplicate cells that appear along axis X at level i and the new axis X' derived from X at level i+1.

Figure 6.2 represents graphically the three types of duplicate cells. The cells with the same color are duplicate cells. Duplicate cells of type 1 and 2 can be removed from the mapping table without having any effect on the cells that will be created afterwards. This is obvious for duplicate cells of type 1. This is because the intersection of two equal cells that belong to the same axis with any other cell will create the same results. However, this is not obvious for duplicate cells that belong to the second category.

\$2 ↑								
max (\$2)								
⋮								
j+1								
j	Y			[i,j]				
j-1								
⋮								
1				X				
	1	···	i-1	i	i+1	···	*max* (\$1)	\$1 →

Table 6.3: Original Mapping Table *M*.

Theorem 6.4.1. *If the same set of tuples appears in different edge cells along different axes, then the entire row (or column) where one of the two cells appears can be eliminated without losing any information.*

Proof. Assume a function *tuples(c)* that returns the set of tuples that appear in cell c. Also assume a mapping table M that has edge cells $X = [i, 1]$ and $Y = [1, j]$ with the property $tuples(X) = tuples(Y)$.

Without loss of generality, I prove that by eliminating the entire row that starts with edge cell Y, the new mapping table, M', contains the same set of slices as M. This proof relies on the fact that the set of tuples maintained by cells preceding a particular cell along the axis is a superset of the tuples maintained by this particular cell. More formally, $tuples([i, 1]) \subseteq tuples([k, 1])$, where $k < i$. Similarly, $tuples([1, j]) \subseteq tuples([1, l])$, where $l < j$. In addition, it relies on the fact that "internal" cells can be calculated using the intersection of edge cells, e.g. $tuples([i, j]) = tuples([i, 1]) \cap tuples([1, j])$.

To prove that M is equivalent to M' the following must be proved:

1. That all the slices appearing in the row that starts with cell Y are redundant.

2. That the elimination of this row does not alter the contents of any other cell.

For proving the first part I utilized the facts that $tuples([i,1]) = tuples([1,j])$ (initial assumption) and also that $tuples([i,1]) \subseteq tuples([k,1])$, where $k < i$. That means that $tuples([1,j]) \cap tuples([k,1])$, where $k < i = tuples([1,j])$. As a result, $tuples([i,1]) = tuples([m,j]$, where $1 \leq m \leq i$.

Above I proved that all cells $[1,j] \cdots [i,j]$ contain redundant information. I should also prove the same for cells [m,j], where $i+1 \leq m \leq max(\$1)$. This can be achieved by induction on m. For the base case, where $m = i+1$, $tuples([i+1,j]) = tuples([i+1,1])$. This is because $tuples([i,1]) = tuples([1,j])$ and $tuples([i+1,1]) \cap tuples([i,1])$. As a result the intersection that returns the $tuples([i+1,j])$ is $tuples([i+1,1]) \cap tuples([1,j]) = tuples([i+1,1])$. For the inductive hypothesis, I assume that $tuples([max(\$1)-1,j]) = tuples([max(\$1)-1,1])$. Using the inductive hypothesis and the same reasoning used above for proving that $tuples([i+1,j]) = tuples([i+1,1])$, one can also prove that $tuples([max(\$1),j]) = tuples([max(\$1),1])$.

The second part of the proof is trivial, given the fact that the tuples contained by a cell is equal to the intersection of tuples contained by edge cells. As long as no tuples are altered in any edge cell, the intersection returns the same results. □

On the contrary, duplicate cells of type 3 cannot be eliminated from the mapping table. To make this clear, I present a counterexample. Consider the mapping tables presented in figure 6.3. In both of these examples, if the duplicate cell <1,5> and <1,4> had not been propagated, the cell <1> would have never

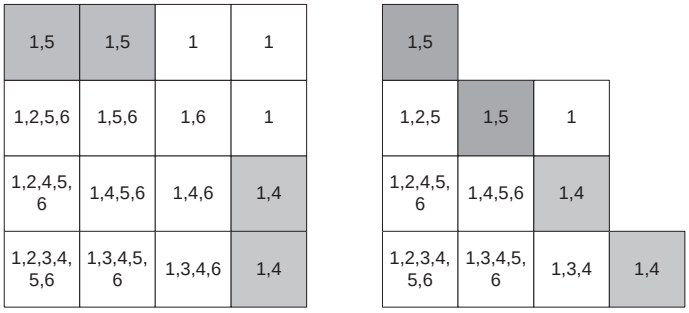

Figure 6.3: Example of Mapping Tables where duplicate cells cannot be removed.

been created. However, duplicate cells that belong to the third category can be marked as "duplicate". In this way, duplicate cells of type 3 are propagated to higher level cells but never saved into the slice file.

6.4.3 Algorithm

Below I present the pseudo-code of the algorithm that creates a mapping table. It assumes that the axes of level 1 can be created by loading a file that contains the necessary information.

```
Mapping Table (dimensions n)
    currentAxes = Create n vectors each representing a different axis at level 1
                  and initialize them by loading the input file.
    Level = 1;
    WHILE(currentAxes.size() > 1)
                  //This process removes any duplicate cell that might exist.
                  RemoveDuplicates(currentAxes);
                  Write cells of currentAxes not marked as duplicate to a file;
                  //Creates the cells of the next level and set the vectors of
```

```
                    //the new level to be the currentAxes.
                    NextLevelAxes(currentAxes);
    END WHILE
END
```

The following function removes duplicate cells that belong to different axes of the active level, i.e., type 2 duplicate cells. That means that if a cell along the Y axis is equal to a cell along the X axis, then one of the two cells will be eliminated. This is achieved by comparing axes 1-2, 1-3, ..., 2-3, 2-4, ..., 3-4, 3-5, ... and so on. Also, if any axis remains with a single element at the end of this function, it is removed and the dimension of the mapping table is reduced by one.

```
RemoveDuplicates(currentAxes)
    FOR(i=0 ; i<n ; i++)
        FOR(j=i ; j<n ; j++)
            //Removes from currentAxes(j) any cells that appear in currentAxes(i).
            removeEqualCells(currentAxes (i), currentAxes (j));
            IF(currentAxes (j).size() == 1)
                remove(currentAxes (j));
                Dimension–;
            END IF
        END FOR
    END FOR
END
```

The NextLevelAxes creates the axes of the next layer by calculating the intersection of the cells of the active level. If any of the new axes has a single element,

then it is eliminated and the dimension of the mapping table is reduced by one.

```
NextLevelAxes(currentAxes)
    nextLevelAxes = new axes of size currentAxes.size();
    //For all axes
    FOR(i=0 ; i<currentAxes.size() ; i++)
       //For each cell of each axis
       FOR(j=0 ; j<currentAxes.axisAt(i).size() ; j++)
          //Calculate the new cell. It is equal to the intersection
          //of currentAxes.axisAt(i).cellAt(j) with all cells at position
          //1 of all other axes (i.e., their position should be k and it
          //should be different than i)
          newCell = currentAxes.axisAt(i).cellAt(j) ⋂
                        currentAxes.axisAt(k≠i).cellAt(1);

          //Check for duplicate cells of type 1.
          //If its duplicate do not propagate.
          IF(newCell is not type 1 duplicate)
             NextLevelAxes(i).add(newCell);
          END IF

          //If the new cell is type 3 duplicate mark it as duplicate.
          IF(newCell is type 3 duplicate)
             newCell.MarkDuplicate;
          END IF
       END FOR
```

```
        IF(NextLevelAxes(i).size() == 1)
            Remove NextLevelAxes(i);
            Dimension–;
        END IF
    END FOR
    CurrentAxes = NextLevelAxes;
END
```

6.5 Experiments

In this section, I present my findings from running the algorithm described above on various examples. The goal was to test the efficiency of the algorithm when the number of tuples and the dimensions increased. For doing that, both artificially created and realistic examples were used. The artificial examples were generated by using a random function. As a result, the values were uniformly distributed.

Figure 6.4 represents the results from running the above algorithm on different examples. These examples were create using the same dimension, but different number of tuples. The results clearly show that when increasing the number of tuples and keeping the number of dimensions fixed, the complexity of the algorithm increases.

Figure 6.5 shows the levels required for the algorithm to converge when the number of tuples is fixed (100 and 500 for chart 100 Tuples and 500 Tuples respectively) and the dimensions of the mapping table change. It is very interesting to observe that by increasing the dimensions of the mapping table, the efficiency of the algorithm also increases. This observation derives from the fact that with

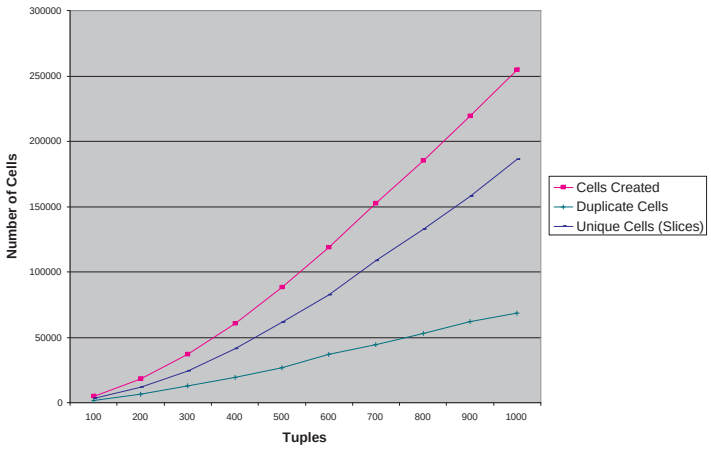

Figure 6.4: Results for 3-dimensional mapping table for different number of tuples.

more dimensions fewer levels are created. This behavior can be explained through figure 6.6.

Figure 6.6 illustrates the number of cells created by the algorithm, compared to the number of unique and duplicate cells generated, when the dimensions of the mapping table increases. Based on the results presented in figure 6.6, one can conclude that when the number of dimensions increases, the total number of cells generated does not change. This is because a mapping table converges faster when it has more dimensions.

In addition, figure 6.6 shows that in higher dimensions more slices are generated and fewer cells are duplicated. This is because the probability of creating duplicate cells of type 3 is reduced, because with more dimensions, a new cell is the result of the intersection of more cells. As a result, the probability that some tuples will be removed and cells that carry different, distinct tuples will be created increases.

Finally, the use of the Mapping Table algorithm on a realistic example showed

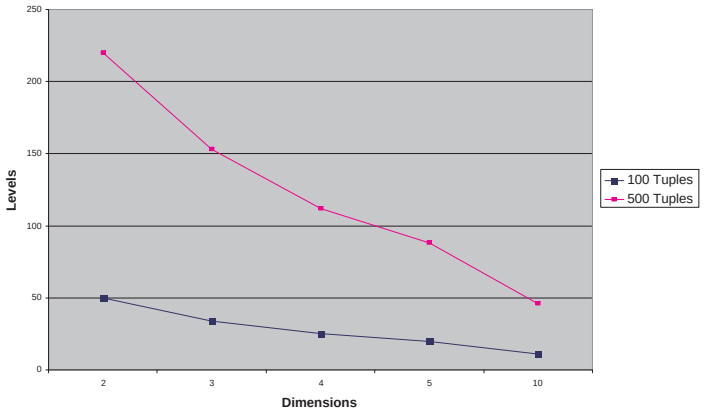

Figure 6.5: Levels required for fixed number of tuples and different dimensions.

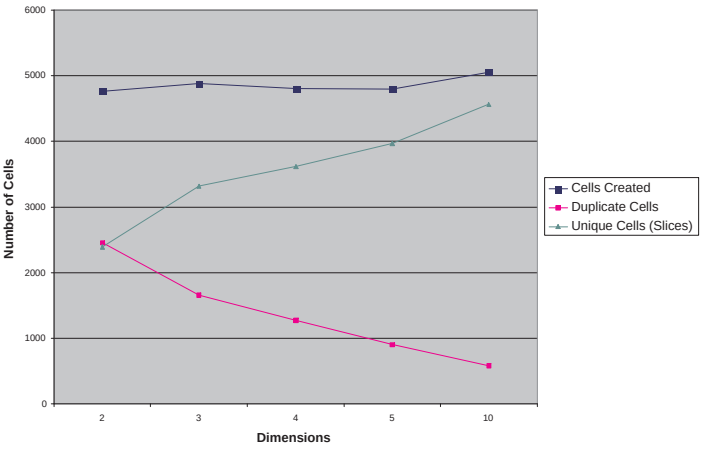

Figure 6.6: Cells Created, Duplicated Cells and Unique Cells.

that the algorithm works even better, when values are not uniformly distributed. This is because in most of the cases, the size of each axis does not increase as the number of tuples increases. Attributes in a database correspond to fields of a network packet. However, in most of the cases, the distinct values that a field can carry does not increase as the number of messages collected increases. For example, in a TCP session the Source IP Address field of any message can only carry two values (the IP address of the two endpoints), regardless of the size of the trace. As a result, creating more tuples by adding more messages does not necessarily mean that the distinct values of a particular attribute will also increase.

6.6 Discussion

In this chapter, a new type of query, the *Parameterized Query* was proposed. A parameterized query derives from an ordinary SQL query by extending the domain of values that one can use for defining it. In addition, a class of *Parameterized Query* that can be computed efficiently was defined. Finally, *Mapping Tables*, an algorithm that efficiently computes the result of *Parameterized Queries* when they belong to a specific class, was proposed. The results suggested that the *Mapping Table* algorithm is extremely efficient and that it scales very well when the dimension of the *Mapping Table* increases.

Chapter 7

Automatic Generation of Automata for TEsting (AGATE)

AGATE is a tool that can be used for creating state machines from a set of network observations. Currently, AGATE creates a state machine either from a raw trace file or from a preprocessed trace file, the *slice file*. The former input file can be created by monitoring the network using a network monitoring tool. For creating the latter input file, the following two steps must be followed:

1. Load a raw trace file into a database.
2. Query the database and save the results of the query into a file, the slice file.

For simplifying this process, a GUI is implemented that assists users to create state machines by using either a trace or a slice file. When starting the GUI the window shown in figure 7.1 appears on the screen. It is divided into two parts. On the left side, there are four buttons: DB Loader, Slicing, SM Gen (Trace) and

SM Gen (Slice), each representing a different functionality that currently AGATE supports. The DB Loader and Slicing can be used for preprocessing a raw trace file, and they correspond respectively to step 1 and step 2 mentioned above. SM Gen (Trace) and SM Gen (Slice) can be used for creating a state machine from a trace and a slice file respectively.

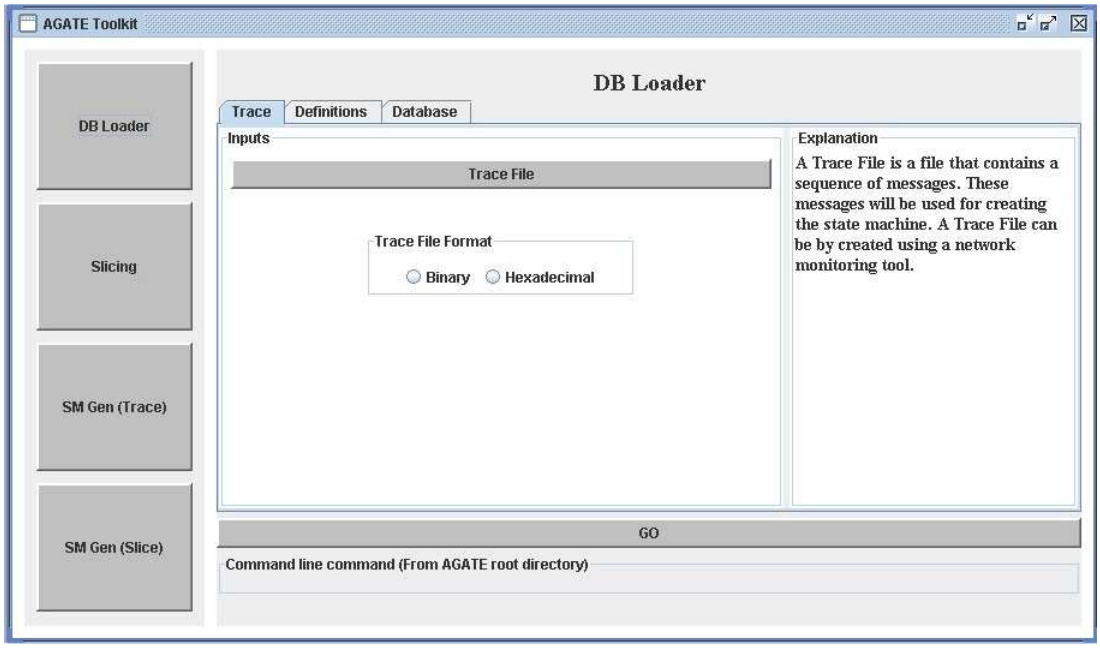

Figure 7.1: AGATE initial window.

The right side is a tabbed panel, and it can be used by the user for providing the necessary inputs or outputs for performing a particular operation. Based on the user's selection (one of the four buttons on the left side), the right side of the panel changes to reflect the necessary inputs and outputs for performing the selected operation.

All the tabs have the same format. They are split into two parts. The left part can be used for specifying input or output information. On the right side, there is a text box that contains an explanation for each of the inputs or outputs.

After providing all the parameters, the selected operation can be performed by clinking the "GO" button, appearing at the bottom of the right panel. In addition to performing the operation, when clicking the "GO" button, a command appears in the "Command-line command" text box. This command can be used as a command-line invocation of AGATE. For using the auto-generated command, the following steps must be followed:

1. Right click once on the text box.
2. Press Ctrl+A on the keyboard for selecting the command.
3. Press Ctrl+C on the keyboard for copying the command.
4. Open a terminal window and go to AGATE's root directory, paste the command, and press Enter.

In the following sections a more detailed description of each of the four functionalities and their inputs and outputs is presented. In addition to this description, the user can refer to the explanation area of each tab.

7.1 Database (DB) Loader

For loading a trace file into a database, the following must be provided: the actual trace file, information for parsing the raw trace file, and information for accessing a particular database. This information can be specified by using the "Trace", "Definitions" and "Database" tabs respectively (figure 7.1):

- Trace: A trace file is a file that contains a sequence of messages. These messages will be used for creating the state machine. A trace file can be created using a network monitoring tool.

- Definitions: The Definitions directory contains definitions of protocols in NetPDL [52] format. These definitions will be used by AGATE for parsing the trace file. Currently, a collection of NetPDL definitions are implemented and included in AGATE.

- Database: This panel contains information about the database where the trace will be loaded. The URL specifies the IP address, name, and (optionally) the table in the database . The *Username* is the username that will be used for accessing the database and the *Password* is the password of the user specified in the *Username* field.

7.2 Slicing

For creating a slice file, the user should provide the following information: a parameterized query, a database previously generated using the DB Loader, and a file for writing the results of the query (figure 7.2). This information can be provided using the "Query", "Database" and "Slice" tabs respectively:

- Query: The file contains a Parameterized Query. The query will be used by AGATE for creating slices of a trace from a table in a database.

- Database: This panel contains information about the database from which the trace will be read. The URL specifies the IP address, name, and the table in the database . The *Username* is the username that will be used for accessing the database, and the *Password* is the password of the user specified in the *Username* field.

- Slice: The Slice Output File contains the slices created by the parameterized query and the database defined in the previous two panels.

Next, I present some additional information about parameterized queries and how they can be defined. A parameterized query is used for slicing a trace file loaded into a database. Each parameter of this query is instantiated using values extracted from the database. For each combination of values extracted from the database, an instantiated query is created. All the messages returned by an instantiated query define a slice.

Example 7.2.1. In the box bellow, I present an example of a parameterized query. The parameterized query starts with the BINARY operation, which is used for assigning direction to messages (i.e., input or output). In this example, messages that carry values 1, 3, 4, 7, or 8 in the *msgType* field are input messages. All other messages are output messages. Finally, two messages belong to the same group, if the values carried by their *xid* and *chaddr* fields are equal.

```
Select
   BINARY (msgType=1 || msgType=3 || msgType=4 ||
               msgType=7||msgType=8) as input, *
FROM table where xid = $1 and chaddr = $2 ;
```

This query creates slices by grouping together all the messages that carry the same value in fields *xid* and *chaddr*. It will first retrieve all the values of *xid* and *chaddr* from DHCPTable. Then, for all the different values retrieved, a different instantiated query will be created and executed on the database. All the records returned by each instantiated query belong to the same slice. □

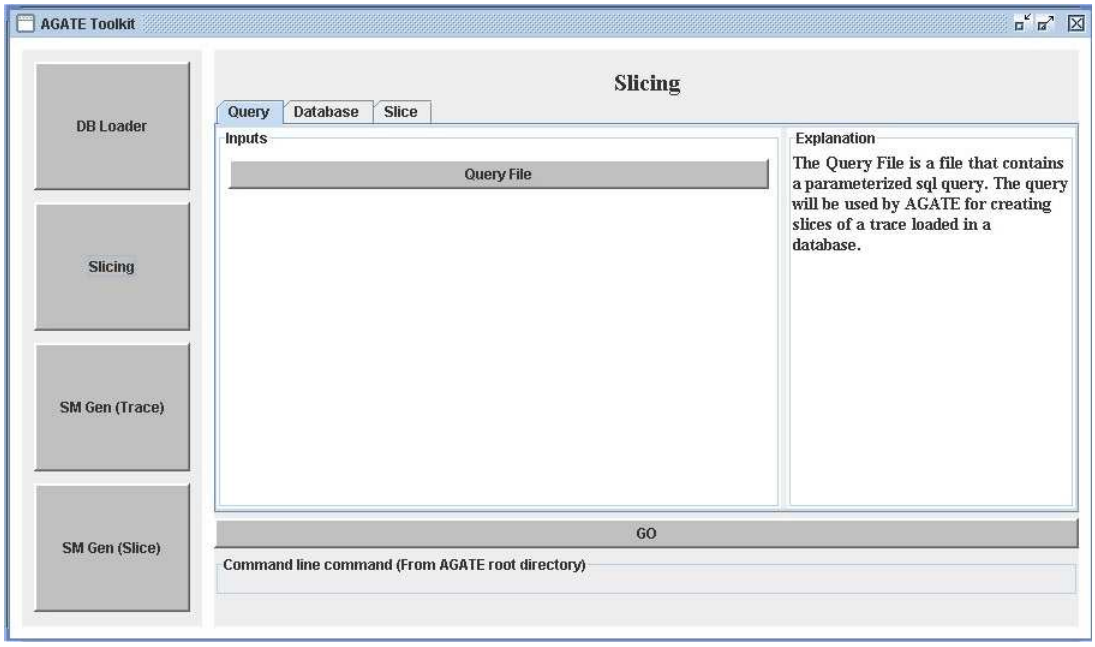

Figure 7.2: AGATE Slicing Window.

7.3 State Machine Generator (SM Gen)

Using this functionality, the user can generate a state machine from either a trace (figure 7.3) or slice (figure 7.4) file. Generating a state machine from a trace file ("SM Gen (Trace)" panel), the user should provide the following information:

- Trace: A Trace File is a file that contains a sequence of messages. These messages will be used for creating the state machine. A Trace File can be created using a network monitoring tool like tcpdump [54] and Wireshark [61].

- Definitions: The Definitions directory contains definitions of protocols in NetPDL [52] format. These definitions will be used by AGATE for parsing the trace file. Currently, a collection of NetPDL definition is provided with AGATE.

- State Machine Definitions (SM Defs): A directory with one or more sub-directories, each one containing enough information for creating state machines. For each sub-directory, one state machine will be created. Each of these sub-directories should contain two files, a .session and a .semantics files (explained bellow).

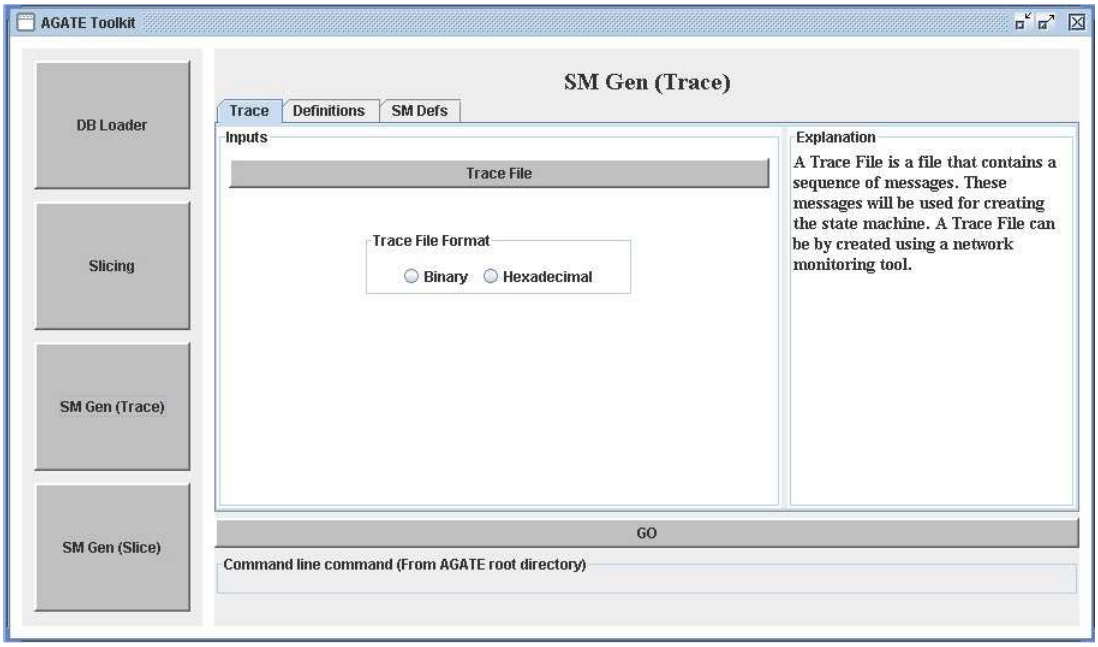

Figure 7.3: State Machine Generator from Traces Panel.

Generating a state machine from a slice file ("SM Gen (Slice)" panel) is similar to the creation of the state machine from a trace file. However, the Definitions tab is missing because messages in a slice file have already been processed and parsed. As a result, this information is redundant when slice files are used.

The SM Defs tab that appears in both versions requires a more detailed explanation. AGATE can create multiple state machines using the same trace file by grouping messages together in different ways and by using different abstractions. This information can be provided to AGATE as a directory structure using the

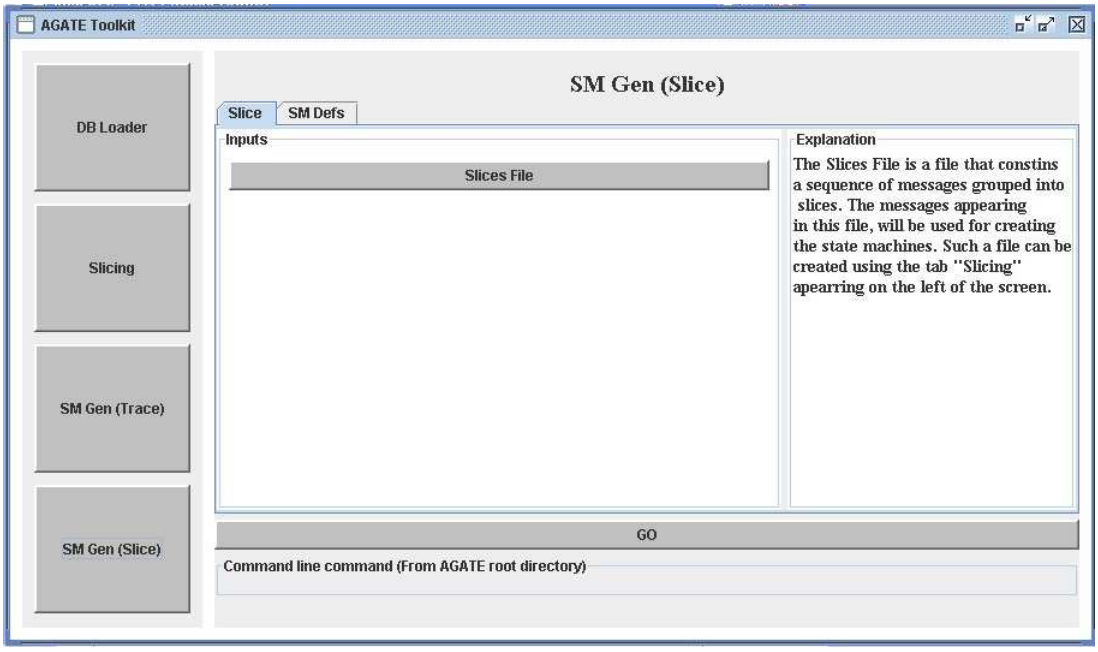

Figure 7.4: State Machine Generator from Slices Panel.

"SM Defs" tab. The directory MUST have one or more sub-directories. For each sub-directory, a different state machine will be created.

When raw trace files are used (figure 7.3), each sub-directory MUST contains two files with extensions ".session" and ".semantics". The ".session" file contains the information for grouping messages together, whereas the ".semantics" carries the abstractions to be used by AGATE.

In the case that slice files are used (figure 7.4), each sub-directory MUST contains one file with extension ".semantics". The ".session" file is not used in this case, because information related to message groups is included in the slice file.

In the next two sections, I describe the format of a ".session" and a ".semantics" file.

7.3.1 Message Grouping

Messages in a trace file must be grouped together in meaningful ways, so that each group describes a permissible behavior of the implementation under test. In AGATE, messages grouped together without using a database are called "sessions", whereas messages grouped together using a database are called "slices". The difference in terminology indicates that in the former case each message is limited to belonging to a single group, while in the latter, the same message mat belong to any number of groups.

One way for grouping messages together is by slicing a trace file through the utilization of a database. This can be done using the functionality described above. However, there are cases where grouping of messages can be performed without the use of a database. The purpose of SM Gen (Trace) is to create state machine in such cases. To do so, it utilizes the ".session" file mentioned above.

Session File (.session): defines the conditions necessary so that two messages belong to the same group.

Format of a .session file: The first line of the file should specify the direction of a message (i.e., input or output). This is used when a TIOA model is created for creating input and output actions (see section 5.1). It is also used by semantics to restrict members of symbolic groups to either input or output messages (see section 4.2). The second line of the file should define the timeout value (in milliseconds) of the session. The timeout is defined as the maximum interval between the time stamp of the last message assigned to the session and the time stamp of a new message that the program attempts to assign to the same session. All other lines in this file describe the relations between fields of the old message (the message that already belongs to a session) and fields of the new message (the message that

the program tries to assign to a session). Each relation has to be defined in CNF form. Each line is considered a clause of disjunctions.

Example 7.3.1. This presents an example of a session file. The first line defines the direction of the message (i.e., input or output). In this example, messages that carry values 1, 3, 4, 7, or 8 in their *msgType* field are input messages. All other messages are output messages. The timeout is defined to be 20 seconds. Finally, two messages belong to the same group, if the values carried by their *xid* and *chaddr* fields are equal.

```
input msgType=1 msgType=3 msgType=4 msgType=7 msgType=8
timeout=20000
new.xid=old.xid
new.chaddr=old.chaddr
```

□

7.3.2 Message Abstraction

Different abstraction is made possible by assigning different semantics to each field (semantics are described in section 4.2). For assigning semantics to fields, AGATE utilizes a semantics file.

Semantics File (.semantics): This file is used for assigning semantics to fields. If the semantics of a field is not specified, then the default value is invisible. Bellow, I present an example of a semantics file that contains all possible assignments currently supported by AGATE.

Example 7.3.2. This is an example of a semantics field. It has not been derived from a particular protocol, however it contains all possible options that one can

use for assigning semantics to fields. Also, have in mind that the default semantic assigned to a field is *Invisible*.

```
//These are symbolic fields
Symbolic field1, field2;

//These are regular fields
Regular field3;

//More symbolic fields
Symbolic field5;

//These are fields that are invisible but for some reason the user
//wants their values be displayed when the state machine will be
//graphically displayed (explained in the next section).
Display field8;

//Commented out line.
//Display field9;

//These are symbolic fields that they belong to the same group.
SymbolicGroup field12, field13;

//These are ordered symbolic fields that they belong to different groups.
OrderedSymbolic field10, field11;

//These are ordered symbolic fields that they belong to the same group.
OrderedSymbolicGroup field12, field13;
```

7.3.3 Generated State Machine

In this section, I present the interface that AGATE uses to graphically represent a state machine. The output frame is split into two panels. The upper panel is used for the graphic representation of the generated state machine. $S0$ represents the initial state of the state machine and all states with double circle line represent final states (S4 in example 7.5). Small circles on the upper right corner of a state

represent self-loops, and edges with red color represent loop-back (go back to a previous state) edges.

The lower panel is used for displaying messages carried by edges, i.e., the label of an edge. When the mouse pointer is passed over an edge, the label of the edge is displayed in this panel. Three different types of fields are displayed in this panel: Regular, Symbolic, Display. More precisely, the actual value of Regular fields, the symbolic value of Symbolic fields, and the actual value of Display fields. Symbolic and Regular fields were described in section 4.2. Display fields are fields that do not participate in the generation of the state machine. However, these are fields that the user considers interesting and wants to know the value they carry.

This interface is also used for translating the generated state machine to a TIOA model. One can do that by clicking on the "TIOA" button, which appears on the top of the frame. An example of a TIOA model created by AGATE is presented in appendix A.

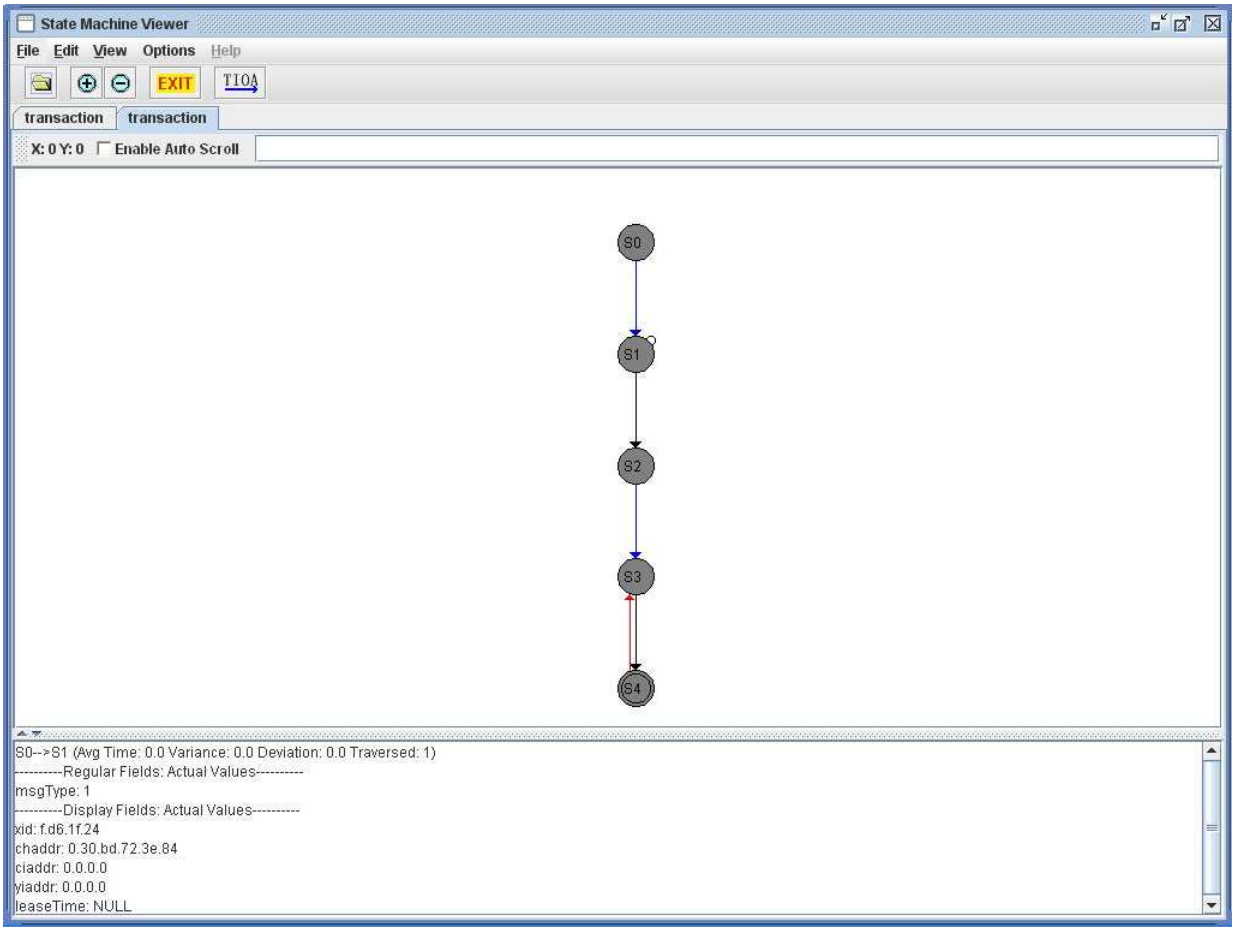

Figure 7.5: An example of a state machine generated by AGATE.

Chapter 8

Addressing Scalability in Network Testing

One of the biggest obstacles of testing is that of scale. In order to partially overcome this obstacle, Djouvas, Griffeth, and Lynch in [19] proposed a way of finding a representative network to test, instead of testing all networks that are members of a class of networks. By finding (if it exists) a sub-network S_N that looks like the entire network N, then the smallest such sub-network is a good candidate for testing. This is because one can test S_N to determine the properties of the entire network N. A network where such a sub-network exists is called *self-similar*.

In this chapter, the formal definition of *self-similarity* is provided. In addition, I describe how *self-similarity* can be established and applied in network testing. An extensive example of using *self-similarity* on a real network implementation is provided in section 8.3.

8.1 Self-Similarity

A *self-similar* network N is a network that has the property that a sub-network S_N of it "looks like" the entire N. Obviously, not all networks has such property. The easiest example of network that trivially does not have such property is the smallest S_N of N.

Definition 8.1.1. A network N is *self-similar*, if a sub-network S_N of N exist, such that $traces(S_N \mid\mid ... \mid\mid S_N) \subseteq traces(N)$ exists.

Definition 8.1.1 provides the basis upon which *self-similarity* is defined using a formal language, TIOA in particular. Because this work is only interested in network testing and assumes that the only realistic approach is black-box testing, only TIOA models with output actions named *send* and input actions named *receive* are considered, i.e., only actions observable by the tester. These automata are parameterized by the number of interfaces they have on the network. Each *send* and *receive* action is associated with one of the interfaces. *Send* action sends the message out of an interface and *receive* action receives a message arriving on an interface.

An automaton with n interfaces has a signature containing at least the actions:

- $send(m : Message, i : Int)$, where $1 \leq i \leq n$,
- $receive(m : Message, i : Int)$, where $1 \leq i \leq n$,

where *Message* is the set of possible messages that can arrive on an interface.

To combine two automata, allowing them to exchange messages, a channel automaton $Channel(a, b)_{i,j}$ is used (described in [47]), which joins interface i

of automaton a to interface j of automaton b. The composed automaton has input actions $send(m,l)_a$ and $send(m,l)_b$ and output actions $receive(m,l)_a$ and $receive(m,l)_b$, where $1 \leq l \leq n$.

Definition 8.1.2. Suppose that n denotes the number of interfaces of an automaton as defined above. Then, an automaton $A(n)$ is self-similar if: $ActHide_{\Phi}$ $(traces(A(n)) \;||\; Channel(a,b)_{i,j} \;||\; traces(A(n))) \subseteq traces(A(2n-2))$, where Φ=$\{send(m,i)_a$,$send(m,j)_b$, $receive(m,i)_a$,$receive(m,j)_b\}$.

In other words, the externally observable actions of the composition of $A(n)$ with itself, using a channel connecting interfaces i and j, looks like a single automaton $A(2n-2)$ missing the interfaces i and j. Definition 8.1.2 can be generalized in the obvious, i.e., by setting up channels and networks with more complex topologies. However, the above definition provides a very basic view of *self-similarity*. In addition, *self-similarity* for properties of networks is defined, since it may be easier to establish *self-similarity* of interesting properties than for entire automata.

Definition 8.1.3. A trace property T is self-similar, if the network $N(n)$ $||$ $Channel_{i,j}$ $||$ $N(n)$ has property T whenever network $N(n)$ has property T.

Results concerning a *self-similar* property of a network $N(n)$ can be generalized to apply to larger networks.

8.2 Self-Similarity in Testing

By the definition of *self-similarity*, a correct behavior of a *self-similar* network N implies that the same behavior will also be correct in any larger network composed of multiple instances of N. Also, bugs in N imply that there are bugs in the larger

network. However, it is easy to observe that not all networks are *self-similar*. Next, I describe two approaches that still allow us to take advantage of *self-similarity* to reduce the size of the network under test N, if N is not *self-similar*. The first approach utilizes a generalized model M of the network N, where M is self-similar and also M is close enough to N to conform to the specification of N. The second approach defines *self-similar* properties of the network and tests for their presence in the smaller network.

8.2.1 Self-Similar using Models

In general, one will not be able to show *self-similarity* of every network that he/she tests. This approach requires a generalized *self-similar* model M of the network N. If the specification S holds on M and if N implements M, one can use the test results as if N itself was *self-similar*. The following theorem is the basis of this claim.

Theorem 8.2.1. *If M is self-similar and if $traces(N) \subseteq traces(M) \subseteq traces(S)$, then $ActHide_{\Phi}\ (traces(N) \mid\mid Channel_{i,j} \mid\mid traces(N)) \subseteq traces(S)$, where $\Phi = \{send(m,i)_a, send(m,j)_b, receive(m,i)_a, receive(m,j)_b\}$.*

This theorem says that given a network N and a *self-similar* model M, where M implements S and N implements M, one can conclude that two composed instances of network N implement S. By induction, any number of instances of N can be composed and still conform to S.

Proof. The theorem follows easily from the properties of self-similarity and composition. $traces(N) \subseteq traces(M)$ implies that $ActHide_{\Phi}\ (traces(N) \mid\mid Channel_{i,j}$

$|| \; traces(N)) \subseteq ActHide_{\Phi} \; (traces(M) \; || \; Channel_{i,j} \; || \; traces(M))$ by the composition theorem [47]. Since M is self-similar, Theorem 8.2.1 follows. □

8.2.2 Self-Similar using Properties

As noted above, test results concerning a *self-similar* property of a network N can be generalized to apply to larger networks.

If *self-similar* trace properties S and T both hold for a network N, then clearly so does the conjunction of S and T $(S \wedge T)$. This can be used to show that if a complex network requires that a number of properties $P_1, ..., P_n$ have to be true, it is enough to show that each of these properties is *self-similar*, rather than trying to show that all are *self-similar* at once.

In general, every property of a network, in which one is interested is not *self-similar*. However, one may be able to show *self-similarity* of a significant subset, so that testing of those properties can be carried out on a smaller network.

Section 8.3.1 presents an example where properties were used for establishing that learning bridges are self-similar.

8.3 Self-Similarity Example

An interesting example of a network implementation that exhibits *self-similar* behavior is that of learning bridges. Learning bridges are layer 2 devices that can be used for connecting different hosts that belong to the same network, allowing them to exchange messages. The intuition behind this example is that learning bridges have a vague idea about their neighbor bridges and cannot distinguish between a port that leads to a single huge bridges with many ports and a collection

of bridges connected with each other that allow many hosts to get connected to the network. Utilizing the *self-similar* property of the learning bridges, the tester can test a single learning bridge for justifying the correctness of a collection of learning bridges connected with each other[1].

In this section, two different approaches that establish that learning bridges are *self-similar* are presented. First, in subsection 8.3.1, axioms that capture the essential properties of a learning bridge are defined and a formal proof that establishes that they are *self-similar* is provided. In subsection 8.3.2, the same results are proved using a model M of a learning bridge defined as TIOA automaton by proving that the composition of two such bridges implements a single bridge. This proof is much more complicated than the property-based proof of subsection 8.3.1, but is extremely interesting because it contains the construction of the actual composed bridge.

8.3.1 Proof Using Self-Similar Properties

For defining the *self-similar* axioms that capture the essential properties of a learning bridge, one must first define the following: Given a set M of messages and an integer n which represents the number of ports that a learning bridge has, define a set of actions $Acts = \{send(m,i),\ receive(m,i)\ | m \in M \wedge 1 \leq i \leq n\}$ and a mapping *dest* from messages $m \in M$ to port i where $1 \leq i \leq n$.

Bridge Axioms:

For any trace of a learning bridge over actions in *Acts*, there is a function f from *send* actions to *receive* actions in the trace, satisfying the following axioms:

[1]Note that here only message forwarding is addressed, i.e., how learning bridges pick the correct port to forward a message, and not the spanning tree protocol.

1. If $f(send(m,i))$=$receive(m,j)$ then $receive(m,j)$ precedes $send(m,i)$ in the trace, i.e., the *receive* action of m should precede the *send* action of m.

2. For each $receive(m,i)$ in the trace, if $i \neq dest(m)$, then there is an action $send(m,\ dest(m))$ in the trace with $f(send(m,dest(m))) = receive(m,i)$, i.e., a message m received by a bridge that is not the final destination of m should eventually be forwarded.

3. f does not map different sends from the same port to the same receive, i.e., each *send* is associated with one *receive*.

4. If $f(send(m,i))$=$receive(m,j)$ then $i \neq j$, i.e., messages cannot be sent out of the port they were received.

5. If $receive(m,i)$=$f(send(m,j))$ precedes $receive(m',i)$=$f(send(m',j))$ in a trace, then $send(m,j)$ precedes $send(m',j)$ in the trace, i.e., the order of messages received on a port is maintained.

Theorem 8.3.1. *Suppose a failure-free network of bridges, connected using universal reliable FIFO channels in a tree configuration. Let $dest_i(m)$ be the destination function for $bridge_i$ and let it satisfy the condition that the unique path from $bridge_i$ to the host that is the destination of m goes out of port $dest_i(m)$. Then, the composition of any pair of bridges, together with the channel (if any) connecting them in this network obeys the above axioms.*

Proof. Let f_1 be the mapping from *send* to *receive* actions in $bridge_1$, and f_2 be the mapping from *send* to *receive* actions in $bridge_2$ and let f_c be the mapping from *receive* to *send* actions in the channel. Let $dest_1$ be the function mapping messages to a port of $bridge_1$ that leads to destination, and let $dest_2$ be the

function mapping messages to a port of $bridge_2$ that leads to the final destination of a message.

If the bridges are not connected to each other in the tree, then $f = f_1 \cup f_2$ satisfies the axioms.

If they are connected to each other, then let i_0 be the port connecting $bridge_1$ to the channel and let j_0 be the port connecting $bridge_2$ to the channel. Let $\Phi = \{send(m, i_0)_1, receive(m, j_0)_2, send(m, j_0)_2, receive(m, i_0)_1\}$ and consider any trace of the composition $ActHide_\Phi$ $(bridge_1 \mid\mid Channel(1, 2) \mid\mid bridge_2)$.

Note that the mapping f_c from the *receive* actions of the $Channel(1, 2)$ to the *send* actions of the channel, as defined in [47] is injective (one-to-one) and surjective (onto).

For purposes of this proof, assume that all messages in M are distinguishable from each other, for example by a sequence number. Thus, $f_c(send(m, i_0)_1) = receive(m, j_0)_2$ and $f_c(send(m, j_0)_2) = receive(m, i_0)_1$.

The claim is that one of the following must hold for the destination functions $dest_1$ and $dest_2$. For each message $m \in M$, either $dest_1(m) = i_0$ or $dest_2(m) = j_0$, but not both. This follows from the fact that the topology is a tree. Thus, one can define a function $dest(m)$ for the composed automata as follows:

1. If $dest_1(m) = i_0$ then $dest(m) = dest_2(m)$.

2. If $dest_2(m) = j_0$ then $dest(m) = dest_1(m)$.

Define a mapping f from external *send* actions of the composition to *receive* actions as follows:

- If $f_1(send(m, i)_1) = receive(m, i')_1$ where $i' \neq i_0$, then $f(send(m, i)_1) = f_1(send(m, i)_1)$.

- If $f_1(send(m,i)_1) = receive(m,i_0)_1$, then $f(send(m,i)_1) = f_2(f_C(receive(m, i_0)_1)) = f_2(send(m,j_0)_2)$.

- If $f_2(send(m,j)_2) = receive(m,j')_2$ where $j' \neq j_0$, then $f(send(m,j)_2) = f_2(send(m,j)_2)$.

- If $f_2(send(m,j)_2) = receive(m,j_0)_2$, then $f(send(m,j)_1) = f_1(f_C(receive(m, j_0)_2)) = f_1(send(m,j_0)_1)$.

The claim is that f satisfies the axioms above.

Axiom 1: Since each function has the property that its image precedes its argument, this follows.

Axiom 2: Let $receive(m,i)_1$ be a *receive* action in a trace of the composite bridge. If $i = dest_1(m)$ there is nothing to prove. Suppose that $receive(m,i)_1$ is an action in the trace with $i \neq dest_1(m)$. Then, by axiom 2, there is an action $send(m,dest(m))_1$ with $f_1(send(m,dest(m))_1) = receive(m,i)_1$. If $dest(m) \neq i_0$, then $f(send(m,dest(m))_1) = f_1(send(m,dest(m))_1)$ which proofs this case. If $dest(m) = i_0$, then m proceeds through $Channel(1,2)$ to $bridge_2$, arriving at port j_0. Since $dest_1(m) = i_0$, $dest_2(m) \neq j_0$, so that there is an action $send(m,j)_2$ with $f_2(send(m,j)_2) = receive(m,j_0)_2$, and $f(send(m,j)_2) = f_1(f_c(f_2(send(m,j)_2)))$ $= f_1(f_c(send(m,j_0)_2)) = f_1(send(m,i_0)_1)$, which by the above is $receive(m,i)$.

Axiom 3: Suppose that *send* actions π_1 and π_2 from the same port are mapped to the same *receive* action. If that action is on a port of the same bridge as the send port, this contradicts the assumption that the axiom holds for the component bridges. Hence π_1 and π_2 correspond to two *receive* actions π_1' and π_2' that arrived at the port joining to the other bridge. Because the channel connecting the bridges never duplicates messages, there are two *send* actions ρ_1 and ρ_2 that happened

at the other end of the channel. By axiom 3, these must be mapped to distinct *receive* actions. By contradiction, axiom 3 holds for the composed bridges.

Axiom 4: If $f(send(m,i)_k)$=$receive(m,j)_k$ then $f_k(send(m,i)_k)$=$receive(m,j)_k$ for $k \in$\{1,2\}, i.e.,, if f maps the send to a receive on the same bridge, then since Axiom 4 holds for f_1 and f_2 and f is equal to one or the other of these. Axiom 4 holds also for f. If f maps the *send* to a *receive* on a different bridge, obviously the *send* and *receive* involve different ports.

Axiom 5: The only cases that has to be proved is when $f(send(m,i)_1) = f_2(f_c(receive(m,i_0)_1)) = f_2(send(m,j_0)_2) = receive(m,j)_2$ and $f(send(m,j)_2) = f_1(f_c(receive(m,j_0)_2)) = f_1(\ send(m,i_0)_1) = receive(m,i)_1$. Suppose $receive(m,i)_1$ precedes $receive(m',i)_1$, then $send(m,i_0)_1$ precedes $send(m',i_0)_1$, because the component automata keep the messages in order by Axiom 5. Also, the channel keeps messages in order, so $receive(m,j_0)_2$ precedes $receive(m',j_0)_2$. By the hypothesis that the component automata obey Axiom 5, this implies that $send(m,j)_2$ precedes $send(m',j)_2$. □

8.3.2 Proof Using a Generalized, Self-Similar Model

The purpose of a learning bridge in a LAN is to deliver messages directly from the source to the destination, while avoiding collisions and duplicated messages. To avoid duplicated messages, the bridge maintains a *filtering database* that maps each destination to the bridge port leading to the destination. Once the bridge has the necessary information in its filtering database, it forwards a message only through the port that leads to the destination.

A network of bridges that conforms exactly to this requirement is not self-similar. Consider the following example: Bridges $bridge_A$ and $bridge_B$ are con-

nected to each other, with $bridge_A$ preceding $bridge_B$ in a path from S (source) to D (destination). Suppose that the filtering database in $bridge_A$ does not contain an entry for D, while the filtering database of $bridge_B$ does contain an entry for D. Then, if a message initiated is from S to D, $bridge_A$ will forward this message to every active port, but $bridge_B$ will forward it only to the correct port. Now suppose that $bridge_{AB}$ is a single bridge derived from the composition of $bridge_A$ and $bridge_B$. If the model included the requirement mentioned above, an external observer would have expected the trace of $bridge_{AB}$ to have only one outgoing message having as destination D. But this will not happen. Instead, the message will be forwarded to all ports that have been inherited by $bridge_A$ and to a single port inherited by $bridge_B$, the same one to which $bridge_B$ would have forwarded the message.

So, a generalized model must be defined, in which it is required that the bridge copies each message to the "correct port", perhaps along with other ports. A "correct port" P is the port at which the last message with source equal to the destination of the current message was received. To implement this, each time a message is received, the learning bridge algorithm records the source address with the port at which the message arrived in the *filtering database*. Subsequent messages sent to that address will be copied to the port. If no message is received from the destination address, the *filtering database* will not have an entry for the address, and the bridge forwards the message to all ports. This generalized model captures precisely the forwarding behavior of connected learning bridges.

8.3.2.1 The Generalized Model

In this subsection, the TIOA model of the generalized bridge is presented. Each bridge has a number of ports and four actions: input action *receive*, output action *send*, and internal actions *copy* and *delete*. It also maintains a filtering database, an input and output buffer for each port, and a tracking array to keep track of where messages have been copied to. The *receive* action adds messages received from other bridges to the designated input buffer of the port at which the message arrives and also updates the *filtering database*. The *send* action sends the first message in the output buffer of a port to the channel to which the port is connected. The *copy* action is responsible for copying messages from input buffers to output buffers. It copies the first message m of an input buffer to an output buffer, without duplicating messages, i.e., it copies m to each output buffer at most once and utilizes the tracking array to avoid duplicate copies. Finally, the *delete* action deletes the first message m from an input buffer. This action is enabled either when m has been copied to at least the "correct" output buffer or m has been copied to all output buffers. An output buffer is "correct" for a message m if its port, based on the *filtering database*, leads to m's destination. Below, a TIOA model of the learning bridge described above is presented.

automaton $bridge_i$**:**

signature
input
$receive(m, inPort)_i$
output
$send(m, outPort)_i$
internal
$copyIn(m, inPort)_i$
$copyOut(m, inPort, outPort)_i$

$delete(m, inPort, outPort)_i$

states

inbuf, an array of input buffers, indexed by {1, ..., n }, one for each port
outbuf, an array of output buffers (FIFO queues) indexed by {1, ..., n}, one for each port, initially all *empty*.
table, an array of FIFO queues indexed by {1, ..., n} × {1, ..., n}, one for each pair of ports, initially all *empty*.
filterDB, a mapping of message destinations to ports of () indexed by {1, ..., n}, initially all *nil*.

transitions

$receive(m, inPort)_i$
effect
add m to $inbuf(inPort)$
set $filterDB(m.src) := inPort$

$send(m, outPort)_i$
precondition
m first element on *outbuf(outPort)*
effect
remove first element from *outbuf(outPort)*

$copyIn(m, inPort)$
precondition
m is the first element on $inbuf[inPort]$
effect
add m to $table[inPort,i]$ for all $i \neq inPort$
remove m from $inbuf[inPort]$

$copyOut(m, inPort, outPort)_i$
precondition
m first element on *table[inPort,outPort]*
effect
add m to *outbuf[outPort]*
remove m from $table[inPort,outPort]$

$delete(m, inPort, outPort)_i$
precondition
m is in the queue $table[inPort,outPort] \wedge$
$filteringdb[dest(m)] \neq nil \wedge filteringdb[dest(m)] \neq outPort$

effect
remove m from $table[inPort,outPort]$

Assume that there are a finite number of active ports in any bridge and that the spanning tree algorithm determines which ports are active.

8.3.2.2 Composition of Bridges

In this subsection, I describe the composition of two bridges. This composition will be used to prove that the learning bridges are self-similar. To do so, one should assume that the Spanning Tree Protocol has been run to completion by all the bridges in the network and that there are no failures. As a result, there is only one active path between any pair of bridges. The convention that port i is a port of $bridge_1$ and j is a port of $bridge_2$ is used.

To show that learning bridges are self-similar, two primitive bridges must be first composed. Let $bridge_1$ and $bridge_2$ be two bridges running the TIOA defined above. Without loss of generality, assume that port i_0 of $bridge_1$ is connected with the port j_0 of $bridge_2$ through $Channel_{i_0,j_0}$, and that these are the only active ports connecting $bridge_1$ and $bridge_2$.

Let $bridge_c$ be the result of composing $bridge_1$ and $bridge_2$ and hiding the $send(m,i_0)_1$ action of $bridge_1$ and the $receive(m,j_0)_2$ action of $bridge_1$, i.e., $bridge_c$ $=ActHide_{\Phi}(bridge_1||Channel_{i_0,j_0}||bridge_2)$, and Φ=$\{send(m,i_0)_1$,$receive(m,j_0)_2\}$.

The goal is to show that $bridge_c$ is essentially the same as a single bridge, which is called $bridge_p$, running the TIOA defined above. This goal requires that $bridge_p$ is able to handle the same number of connections as $bridge_1$ and $bridge_2$ can handle together. The number of ports of $bridge_p$ is two fewer than

the number of ports of $bridge_1$ and $bridge_2$ combined, since two ports are used for the "internal" communication. Thus, if $bridge_1$ and $bridge_2$ have n active ports each, $bridge_p$ has 2n-2 active ports. Also, $bridge_p$ is defined so that port $i \in ports_p$, where $1 \leq i \leq n$, is connected to the same channel as the corresponding port $i \in ports_1 - i_0$ of $bridge_1$. Similarly port $j \in ports_p$, where $n \leq j \leq 2n$, is connected to the same channel as the corresponding port $j \in ports_2 - j_0$ of $bridge_2$. Finally, the input and output actions of $bridge_p$ are renamed so that the actions on port i, $1 \leq i \leq n$, are $receive(m, i)_1$ and $send(m, i)_1$; similarly, actions on port j, $n \leq j \leq 2n$, are $receive(m, j - n)_2$ and $send(m, j - n)_2$.

8.3.2.3 Simulating a bridge with a composition of bridges

An important theorem about TIOA is used to show the equivalence of the composed $bridge_c$ to the single bridge $bridge_p$. The theorem says that if there is a simulation relation (definition 2.5.2) from a TIOA A to a TIOA B, then $traces(A) \subseteq traces(B)$. In this subsection, a relation from $bridge_c$ to $bridge_p$ is defined and a proof that is a simulation relation is provided. The pair (s, t) with $s \in states(bridge_c)$ and $t \in states(bridge_p)$ belongs to the relation R, provided that the following conditions hold:

Condition 1: The filtering database of $t.bridge_p$[2] contains the same entries as the union of the filtering databases of $s.bridge_1$ and $s.bridge_2$, excluding the entries for the "internal" ports: $t.bridge_p.filterDB = s.bridge_1.filterDB \cup s.bridge_2.filterDB$ - $\{\langle addr, port\rangle \mid port \in s.bridge_1.i_0, s.bridge_2.j_0\}$

Condition 2: The output buffer for each port of $t.bridge_p$ contains the same

[2]The dot notation is used to denote the value of a given variable in a state as well as to denote a given bridge in the composition.

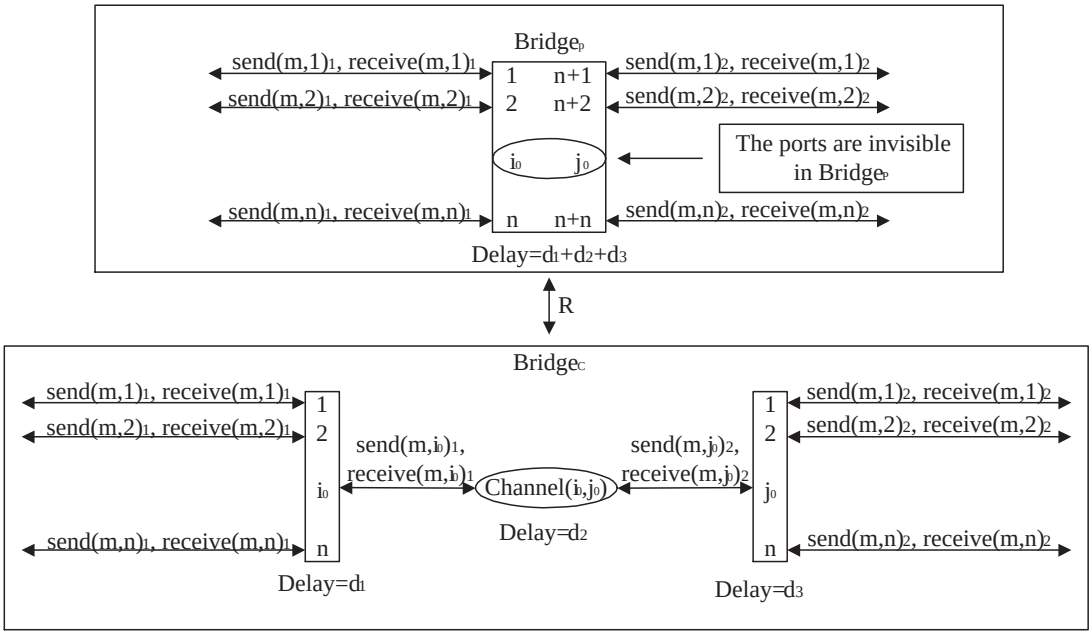

Figure 8.1: Composition and Simulation Relation.

messages as the output buffer of the corresponding port of $s.bridge_c$: $t.bridge_p.outbuf[i] = s.bridge_c.outbuf[i]$ for $i \in ports_1 \cup ports_2$ - $\{i_0, j_0\}$. (Note that $t.bridge_p$ does not contain any buffers corresponding to i_0 and j_0. These buffers in $s.bridge_c$ may contain any messages consistent with the next condition.)

Condition 3: The input buffer for each port of $t.bridge_p$ contains the same messages as the input buffer of the corresponding port of $s.bridge_c$: $t.bridge_p.inbuf[i] = s.bridge_c.inbuf[i]$ for $i \in ports_1 \cup ports_2$ - $\{i_0, j_0\}$.

Condition 4: The internal table of message queues $table_p$ corresponds to the combined tables $table_1$ and $table_2$ as follows:

- $table[i, i']_p = table[i, i']_1$ if $i, i' \in ports_1$ and $i, i' \neq i_0$.
- $table[j, j']_p = table[j, j']_2$ if $j, j' \in ports_2$ and $j, j' \neq j_0$.
- $table[i, j]_p$ is the concatenation of the following queues for $i \in ports_1$, $j \in$

$ports_2$ with $i \neq i_0, j \neq j_0$:

- $table[j_0, j]_2$
- $outbuf[j_0]_2$
- $queue_{j_0,i_0}$
- $inbuf[i_0]_1$
- $table[i, i_0]_1$

- The relationship between $table[j, i]_p$ and $table_1$ and $table_2$ is defined symmetrically for $i \in ports_1$, $j \in ports_2$ with $i \neq i_o$, $j \neq j_o$.

8.3.2.4 Self-Similarity of the Generalized Bridge Model

Self-similarity requires that $traces(M||M) \subseteq traces(M)$, i.e.,, $traces(bridge_c) \subseteq traces(\ bridge_p)$. Based on the definition of simulation relation, two conditions must be proved: the start condition and the step condition. The former is trivial because all the states of both bridges are initially empty. The latter condition requires the proof that the states of $bridge_p$ and $bridge_c$ correspond after each action. First, the proof that establishes state correspondence for the filtering databases is presented.

Invariant 8.3.2. *At any time during the execution, the filtering database of $bridge_c$ corresponds to the state of the filtering database of $bridge_p$.*

Proof. Invariant 8.3.2 can be proved using induction on the number of steps.

Base: At the beginning, everything is empty so the state of the filtering database of $bridge_c$ corresponds to the state of the filtering database of $bridge_p$.

Inductive Hypothesis: Suppose that the invariant holds after t steps.

Inductive Step: Suppose that a message from source s is received at the beginning of step t+1. (If nothing is received, no changes will be made, so the filtering databases remain in corresponding states.) Assume also that the message is received at a port of $bridge_1$ (the case for a message received at $bridge_2$ is symmetric).

There are four different cases:

- Only the filtering database inherited from $bridge_1$ has an entry for the source.
- Only the filtering inherited from $bridge_2$ has an entry for the source.
- Both have an entry.
- Neither has an entry.

First, have in mind that entries in the filtering databases never change (again once they are non-null), given the assumption that there are no failures in the network and that the active ports form a tree. This is the case, because there is a unique path between every pair of ports. As a result, a message from each source to each destination always follows the same unique path.

In the first case, $bridge_c$ already has an entry in the filtering database (for $bridge_1$), and so the state is unchanged. $bridge_p$ also has an entry already (based on the inductive hypothesis), and so changes nothing to its filtering database, so the databases remain in corresponding state.

For the second case, the $bridge_c$ entry is in the filtering database for $bridge_2$, but the message arrives at a port inherited from $bridge_1$. Again, because the network is a tree and only $bridge_2$ has an entry in its filtering database about the source of a message received (not $bridge_1$), the message must have come to

$bridge_1$ from $bridge_2$. Thus, the $bridge_1$ port at which the message arrives is i_0 (the port connected to $bridge_2$) and so the filtering database of $bridge_1$ will point to $bridge_2$. In $bridge_p$ the "internal" entries, i.e., entries pointing from $bridge_1$ to $bridge_2$ and vice versa, are omitted. As a result, the correspondence between $bridge_p$ and $bridge_c$ is unchanged.

In case 3, both bridges have an entry in their filtering databases. Since entries never change, the filtering databases remain in corresponding states.

In case 4, neither database has an entry (hence the message cannot have arrived on the port i_0 connected to $bridge_2$). $bridge_p$ will set the entry for the source in its filtering database to the arrival port. Similarly, $bridge_c$ will set the entry in the filtering database for $bridge_1$ to the arrival port. Thus, the databases maintain their correspondence. □

One must also prove that the states of $bridge_p$ and $bridge_c$ correspond after each action despite changes to input and output buffers. In order to do that, one must consider all the cases based on all actions π that can be taken. The following table (Table 8.1) summarizes all the possible actions of $bridge_c$, the corresponding execution fragment of $bridge_p$ and the trace which is the same for both bridges.

Proof. The goal is to show that if $bridge_c$ is in state s and $bridge_p$ is in state t and if $t \in f(s)$ then for any action π of $bridge_c$, with $(s, \pi, s') \in transitions(bridge_c)$, the corresponding execution fragment of $bridge_p$, given by Table 8.1, takes $bridge_p$ to a state t' such that $t' \in f(s')$.

Line 1: $\pi = receive(m, i)_1, i \neq i_0$. This action adds a message to $inbuf[i]_1$ in $bridge_1$ of $bridge_c$. The corresponding action (also $receive(m, i)_1$) in $bridge_p$ adds a message to $inbuf[i]_p$. Thus the messages in the input buffers remain the same, so $t' \in f(s')$.

	Action of $bridge_c$	Execution fragment of $bridge_p$	Trace
1	$receive(m,i)_1, i \neq i_0$	$receive(m,i)_1$	$receive(m,i)_1$
2	$receive(m,j)_2, j \neq j_0$	$receive(m,j)_2$	$receive(m,j)_2$
3	$receive(m,i_0)_1$	λ	λ
4	$receive(m,j_0)_2$	λ	λ
5	$send(m,i)_1, i \neq i_0$	$send(m,i)_1$	$send(m,i)_1$
6	$send(m,j)_2, j \neq j_0$	$send(m,j)_2$	$send(m,j)_2$
7	$send(m,i_0)_1$	λ	λ
8	$send(m,j_0)_2$	λ	λ
9	$delete(m,i,i')_1, i' \neq i_0$	$delete(m,i,i')_1$	λ
10	$delete(m,j,j')_2, j' \neq j_0$	$delete(m,j,j')_2$	λ
11	$delete(m,i,i_0)_1$	Sequence $delete(m,i,j)_p$ for $j \in ports_2, j \neq j_0$	λ
12	$delete(m,j,j_0)_2$	Sequence $delete(m,j,i)_p$ for $i \in ports_1, i \neq i_0$	λ
13	$copyIn(m,i)_1, i \neq i_0$	$copyIn(m,i)_p$	λ
14	$copyIn(m,j)_2, j \neq j_0$	$copyIn(m,j)_p$	λ
15	$copyIn(m,i_0)_1$	λ	λ
16	$copyIn(m,j_0)_2$	λ	λ
17	$copyOut(m,i,i')_1, i' \neq i_0$	$copyOut(m,i,i')_p$	λ
18	$copyOut(m,j,j')_2, j' \neq j_0$	$copyOut(m,j,j')_p$	λ
19	$copyOut(m,i,i_0)_1$	λ	λ
20	$copyOut(m,j,j_0)_1$	λ	λ

Table 8.1: Correspondence between actions of $bridge_c$ and $bridge_p$

Line 2: $\pi = receive(m,j)_2, j \neq j_0$. The reasoning is the same as for Line 1.

Line 3: $\pi = receive(m,i_0)_1$. This removes a message m from the head of $queue_{i_0,j_0}$ and adds it to the end of $inbuf[i_0]_1$. This could affect the relationship between $table[\ filterDB(m.src)i]_p$ and the various queues in the composition, since the concatenation involves $inbuf[i_0]_1$ and $queue(i_0,j_0)$. However, for all $i \in ports_1$, this leaves unchanged the value of the concatenation: $table[i_0,i]_1 \frown inbuf[i_0]_1 \frown queue_{j_0,i_0} \frown outbuf[j_0]_2 \frown table[filterDB(m.src)_2,j_0]_1$ it follows for all $t \in f(s)$ that $t \in f(s')$. This justifies the choice of λ, which doesn't change

the state t, as the corresponding execution fragment in $bridge_p$.

Line 4: $\pi = receive(m, j_0)_2$. The reasoning is the symmetric to line 3.

Line 5: $\pi = send(m, i)_1, i \neq i_0$. This action removes the message m from the queue $outbuf[i]_1$ in $bridge_1$ of $bridge_c$. The corresponding action in $bridge_p$ is also $send(m, i)_1$, which also removes m from $outbuf[i]_p$. Thus the state $t' \in f(s')$.

Line 6: $\pi = send(m, j)_2, j \neq j_0$.The reasoning is symmetric to Line 5.

Line 7: $\pi = send(m, i_0)_1$. This action removes a message m from $outbuf[i_0]_1$ and adds it to $queue_{i_0,j_0}$ in the composition bridge. As with the action in Line 3, this leaves the concatenation of queues between $bridge_1$ and $bridge_2$ unchanged, so that if s' is the resulting state and $t \in f(s)$ then $t \in f(s')$.

Line 8: $\pi = send(m, j_0)_2$. The reasoning is symmetric to Line 7.

Line 9: $\pi = delete(m, i, i')_1, i' \neq i_0$. This action removes m from $queue[i, i']_1$. If $i \neq i_0$, this entry is related by the simulation relation to the corresponding entry of $queue[i, i']$ of $bridge_p$ and so the corresponding execution fragment is just $delete(m, i, i')_p$, which removes the same entry from $queue[i, i']_p$.

Line 10: $\pi = delete(m, j, j')_2, j' \neq j_0$. The reasoning is the same as in Line 9.

Line 11: $\pi = delete(m, i, i_0)_1$. This removes the message m from $table[i, i_0]$. This queue is part of every concatenation of queues of the form: $table[j, j_0]_2 \frown inbuf[j_0]_2 \frown queue_{i_0,j_0} \frown outbuf[i_0]_1 \frown table[i, i_0]_1$. In other words, the corresponding queues in $bridge_p$ are the queues $table[i, j]_p$ for every $j \in ports_2$ with $j \neq j_0$. Thus any sequence of deletes α that removes m from all queues $table[i, j]_p$ with $j \in ports_2$ and $j \neq j_0$ will leave each of these queues in correspondence with the above concatenation, i.e.,, if $(s, delete(m, i, i_0)_1, s')$ is a transition of $bridge_c$, then (t, α, t') is a transition of $bridge_p$ and $t' \in f(s')$.

Line 12: $\pi = delete(m, j, j_0)_2$. The argument is symmetric to the argument

for Line 11.

Line 13: $\pi = copyIn(m,i)_1, i \neq i_0$. The result of this action is to put m in all queues $table[i,i']_1, i' \neq i$. The action $copyIn(m,i)_p$ does the same to the corresponding queues $table[i,i']_p$ for $i' \in ports_1$ and $i' \neq i_0$. The case of $i' \neq i_0$ is more interesting. Adding m to the queue $table[i,i_0]_1$ adds m to all concatenations of the form $table[j,j_0]_2 \cap inbuf[j_0]_2 \cap queue_{i_0,i_0} \cap outbuf[i_0]_1 \cap table[i,i_0]_1$, where $j \in ports_2, j \neq j_0$. But also, the action $copyIn(m,i)_p$ adds m to all queues $table[i,j]_p$ with $j \in ports_2, j \neq j_0$. So the simulation relation is preserved by these actions.

Line 14: $\pi = copyIn(m,j)_2, j \neq j_0$. The argument is symmetric to Line 13.

Line 15: $\pi = copyIn(m,i_0)_1$. This removes m from $inbuf[i_0]_1$ in $bridge_1$ and adds it to $table[i_0,j]_1$ for all $j \neq i_0$. This does not change the value of any of the concatenations, and so using the corresponding execution fragment α preserves the simulation relation.

Line 16: $\pi = copyIn(m,j_0)_2$. The argument is symmetric to Line 15.

Line 17: $\pi = copyOut(m,i,i')_1, i' \neq i_0$. This action removes m from $table[i,i']_1$ and adds it to $outbuf[i']_1$. The same effects are achieved in $bridge_p$ by applying $copyOut(m,i,i')_p$.

Line 18: $\pi = copyOut(m,j,j')_2, j' \neq j_0$. The argument is the same as in Line 17.

Line 19: $\pi = copyOut(m,i,i_0)_1$. This action removes m from $table[i,i_0]_1$ and adds it to $outbuf[i_0]_1$. This has no effect on the value of any of the relevant concatenations, and so the empty execution fragment in $bridge_p$ preserves the simulation relation.

Line 20: $\pi = copyOut(m,j,j_0)_1$. The argument is symmetric to Line 19. Based

on the above one can conclude that $traces(bridge_p) \subseteq traces(bridge_c)$. □

Chapter 9

Conclusions and Future Work

This work proposed a novel technique that can be used for testing network systems. Its goal was to improve the quality of network testing while reducing the time and effort best practices in network testing require. More precisely, a technique that automatically creates a model that represents the Implementation Under Test (IUT) through the utilization of observations acquired by monitoring its behavior was proposed. In this way, the tester avoids the time consuming process of creating models by hand, a step required by previous network testing techniques. Having a model that represents the IUT, one can prove the correctness of the IUT by proving the correctness of the generated model.

9.1 Summary of Main Results

The proposed network testing approach mimics experimental methods. For building a model that represents the IUT, repeated rounds of "experiments" are required, so that each round acquires deeper knowledge about the implementation.

The main focus of this thesis is the automatic generation of models that contain the maximum number of behaviors using a single experiment, i.e., a set of observations. The process of building a model of the implementation from observations was split into four steps:

1. Analyze and break each observed sequence of messages into smaller subsequences, each capturing a permissible behavior of a component (or a collection of components) of the implementation. Each subsequent represents a permissible word of the generated model.

2. Abstract each word created in step 1 to allow the creation of more complete models.

3. Build a state machine using the abstracted words created in step 2.

4. Minimize the state machine created in step 3 by both identifying and adding loops and also performing classical minimization techniques [2].

In chapter 6, I presented an approach that creates arbitrary subsequences of the original sequence of messages observed. It utilizes existing database technology and an extended form of a relational algebra query, the *Parameterized Query*. Parameterized queries are relational algebra queries allowed to have parameters in place of attributes or domain values. By instantiating the parameters of a parameterized query, an ordinary relational algebra query is created and can be applied on a database. Tuples returned by each instantiated parameterized query define a subsequence of the original sequence. Each subsequence created is consider a word that the generated model will accept. The main drawback of this approach is its complexity. This is because the total number of instantiated parameterized

queries is exponential to the number of the parameters of a parameterized query. To overcome this difficulty, a class of *Parameterized Queries* that facilitate efficient calculation of their results was defined. In addition, an algorithm called *Mapping Tables* that efficiently calculates the results of this class of parameterized queries was proposed. Results suggested that the *Mapping Table* algorithm is extremely efficient and that it scales very well when the number of parameters in a parameterized query increases.

In addition, techniques that automatically identify and add loops to the generated model were proposed and implemented. By adding loops to a model both the number of its states was reduced and its completeness was increased. The proposed algorithm was tested with a number of examples. Results showed that the algorithm efficiently and effectively identified and added loops to models.

The procedure for creating models from observations described above was implemented as part of Automatic Generation of Automata for TEsting (AGATE)[1], an open-source software tool that creates models represented as state machines from a set of observations. In addition, AGATE was equipped with a translator, that translated the state machine into a formal language, TIOA [22] in particular. Tempo [57] toolkit was then used for translating the TIOA model to either UPPAAL [39] for model checking or TAME [9, 35] for theorem proving.

An advantage of the above approach is that it can be used for creating models of individual components of the IUT. Each model should reflect the functionality of the IUT related to the property that the tester wants to prove. For example, in TCP, among others, one can isolate two components, one that handles the reliable delivery of messages (i.e., retransmissions) and one responsible for congestion

[1] AGATE is presented in section 7.

control. Creating models of individual components of the IUT, and not modeling the IUT as a whole, allows the creation of smaller models, which facilitates easier proofs.

For proving the concept of building models described above, AGATE was used for creating the models of three different protocols. More precisely, I created a DHCP [3] model that captures the component of a DHCP server that keeps track of the availability of a particular IP address; a TCP [33] model that handles retransmissions; and a STP [31] model that elects a leader bridge in a bridged network (more detailed explanation of these models was presented in section 5.2). With only a few observations, AGATE was able to create models capturing a substantial part of the functionality of the component of the IUT under investigation. As a result, despite the fact that the proposed approach is closely related to the exponential problem of machine identification [49], using the proposed heuristics AGATE was able to create representative models efficiently and effectively.

Using the three models described above, I also tried to prove that they satisfy some properties using theorem proving, PVS [51] in particular. For simplifying the proving process, I collaborated with Myla Archer and Elizabeth Leonard from the Center for High Assurance Computer Systems (CHACS) of the Naval Research Laboratory (NRL). I utilized the reference variable approach proposed by Leonard and Archer in [43] and also an approach I proposed that creates auxiliary lemmas that can be utilized for making proofs. I applied these two approaches to all three models described earlier. Results suggested that one can define classes of models amenable to more efficient proofs. One such class is presented in section 5.2.

Finally, I presented an approach that overcomes the obstacle of scale, i.e., not all possible topologies and configurations can be tested in a lab, in specific

networks. More precisely, I presented a way of finding a representative network in a class of networks, so that by proving the correctness of the representative network, one also proves the correctness of any member of its class. To justify this approach, a formal proof that learning bridges are self-similar was presented. The complete proof is included in section 8.3.

9.2 Future Work

Avenues for further research include many extensions and generalizations of the work accomplished and described in this thesis. First is the augmentation of the model through the utilization of new observations acquired by new experiments. Efficient approaches should utilize information acquired from previous experiments, so that they avoid experiments that do not add any new information to the model. One approach for achieving that is the implementation of a test agent responsible for creating new experiments. The test agent will be equipped with the partial model of the implementation generated so far, and a set of values that fields of messages can get. The test agent should randomly probe the implementation with new messages, creating new experiments, which will be used for expanding the model.

A test agent can also be equipped with a set of properties that must hold on the implementation. In that case, the test agent should conduct experiments having as a goal to force the implementation to violate any of these properties. A different approach that can be used when a test agent knows the properties that must hold on the implementation is to use techniques described in [23] for creating an upper bound of all legal behaviors of the implementation. Having

such a model, the test agent should collect enough information for pruning the upper bound model by probing the implementation and thus create a model that reflects the actual implementation.

Future research should also focus on techniques that estimate the completeness of the generated model. Measuring the model's completeness is a difficult problem and closely related to the problem of finding correct *stopping conditions*, i.e., when testing should stop. Based on Myers [50], good stopping conditions are heavily based on the experience and the intuition of the tester. This suggests that an experienced tester should be able to "estimate" the completeness of a model. Also, the completeness of the generated model can be estimated by the test agent. Since the test agent knows the number of different messages sent to the implementation, the percentages of those messages that got acknowledged (i.e., it received a response from the implementation) and the number of all possible messages, it can estimate the completeness of the model.

Further investigation should also be conducted on the integration of time in the generated model. The current implementation collects the set of elapsed times between messages in the process of building the session state machine. This allows to perform statistical analysis related to possible times that an event can happen (i.e., when a message can be sent or received). Based on the experience gained from the different protocols generated using this approach, models contained enough information for defining and proving time related properties. However, examples that heavily rely on time should be investigated and, if necessary, extend the current notion of time.

Approaches that improve the effectiveness of the algorithm that recognizes loops can also be investigated. The approach defined in section 4.4 that automat-

ically recognizes and adds loops to a model is a general approach that can be used with any model. However, in networks, the majority of loops (i.e., retransmissions of messages) are triggered by timeouts. As a result, the variance of the elapsed time between messages that belong to the same loop should be small. This suggests that the algorithm proposed in section 4.4 should be extended to take into consideration the elapsed time between messages when it recognizes loops. By doing that, the effectiveness of the algorithm can be improved even further.

Finally, further techniques must be proposed that simplify proofs when theorem proving is used. In section 5.2, two such approaches were utilized with great results. However, by utilizing the fact that all models generated by AGATE share similar structure, I strongly believe that more efficient strategies can be implemented. Efficient strategies, specifically designed for models generated by AGATE that represent network implementations, can dramatically simplify proofs. This is because most of the steps of the proof will be automatically proved by a strategy and not by the tester.

Appendix A

From State Machines to TIOA (Complete Model)

This appendix presents the complete TIOA model presented in section 3.1.5.

```
%This TIOA model is compatible with Tempo Version: 0.1.9 Beta

%This defines a message based on the message definition file
vocabulary packetVocab types

  % The type "Direction" indicates the direction of the message (in or out)
  Direction: Enumeration [inMsg, outMsg],

  % The type "Address" contains some number of octets, each represented by an Int
  Address: Array [Int, Int],

  % The message type contains the time, direction, and fields of the protocol
  Message: Tuple[
    dir: Direction,

    % Regular Fields
    msgType: Int,
```

```
    % Symbolic Fields
    % Symbolic Fields are represented as Int

    % xid is the transaction id
    xid: Int,
    % chaddr is the client id
    chaddr: Int
  ]
end
% End of vocabulary packetVocab

%This defines a timed message queue.
vocabulary timed_queue
  imports packetVocab
  types Message, timed_message_Queue, timed_message
  operators
    mtQ: -> timed_message_Queue,
    enQ_qn: timed_message_Queue -> Bool,
    deQ: timed_message_Queue -> timed_message_Queue,
    enQ: timed_message, timed_message_Queue -> timed_message_Queue,
    MKtimed_message: Message, Real -> timed_message,
    earliest_msg: timed_message_Queue -> Message,
    earliest_deadline: timed_message_Queue -> AugmentedReal,
    latest_deadline: timed_message_Queue -> Real,
    time_ordered: timed_message_Queue -> Bool,
    nthQ: timed_message_Queue, Nat -> Message,
    lengthQ: timed_message_Queue -> Nat,
    isinQ: timed_message, timed_message_Queue -> Bool,
    deadline: timed_message -> Real,
    message: timed_message -> Message
end

% The automaton
automaton communication

imports packetVocab, timed_queue

signature
  input receive(m:Message)
  output send(m:Message)
```

```
states
   curState: Int := 0;
   queue: timed_message_Queue := mtQ;
   clock: Real :=0;
   delay: Real :=0;

transitions

   %Input Actions
   input receive(m) where m.msgType = D ∧
                          m.xid = 1 ∧
                          m.chaddr = 1 ∧
                          clock = 0
     eff
       if(curState = 0) then
          queue := enQ(MKtimed_message(m,clock), queue);
          curState := 1;
          delay := 0;
       fi;

   input receive(m) where m.msgType = D ∧
                          m.xid = 1 ∧
                          m.chaddr = 1 ∧
                          clock = delay + 1
     eff
       if(curState = 1) then
          queue := enQ(MKtimed_message(m,clock), queue);
          curState := 1;
          delay := delay + 1;
       fi;

   input receive(m) where m.msgType = R ∧
                          m.xid = 1 ∧
                          m.chaddr = 1 ∧
                          clock = delay + 2
     eff
       if(curState = 2) then
          queue := enQ(MKtimed_message(m,clock), queue);
          curState := 3;
          delay := delay + 2;
       fi;
```

```
input receive(m) where m.msgType = Re ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = delay + 100
  eff
    if(curState = 4) then
       queue := enQ(MKtimed_message(m,clock), queue);
       curState := 5;
       delay := delay + 100;
    fi;

input receive(m) where m.msgType = Re ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = delay + 100
  eff
    if(curState = 6) then
       queue := enQ(MKtimed_message(m,clock), queue);
       curState := 5;
       delay := delay + 100;
    fi;

%Output Actions
output send(m) where m.msgType = O ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = delay + 5
  pre
    curState = 1;
  eff
    queue := enQ(MKtimed_message(m,clock), queue);
    curState := 2;
    delay := delay + 5;

output send(m) where m.msgType = A ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = delay + 6
  pre
    curState = 3;
```

```
    eff
      queue := enQ(MKtimed_message(m,clock), queue);
      curState := 4;
      delay := delay + 6;

  output send(m) where m.msgType = A ∧
                       m.xid = 1 ∧
                       m.chaddr = 1 ∧
                       clock = delay + 6
    pre
      curState = 5;
    eff
      queue := enQ(MKtimed_message(m,clock), queue);
      curState := 6;
      delay := delay + 6;

  %Trajectories
  trajectories
    trajdef traj
      evolve
        d(clock) = 0.001;
```

Bibliography

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In *SIGCOMM*, pages 118 – 125, Philadelphia, 1990. 2, 17

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. 37, 42, 44, 155

[3] S. Alexander and R. Droms. DHCP options and BOOTP vendor extensions, 1993. 81, 157

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. 42

[5] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. 28

[6] M. Archer, C. Heitmeyer, and E. Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *FMSP '00: Proceedings of the third workshop on Formal methods in software practice*, pages 25–36, New York, NY, USA, 2000. ACM. 28

[7] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9:201–232, 2002. 28, 78

[8] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *UITP '98*, July 1998. 28, 78

[9] M. Archer, H. Lim, N. Lynch, S. Mitra, and S. Umeno. Specifying and proving properties of timed I/O automata in the tioa toolkit. *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 129–138, 27-30 July 2006. 25, 73, 156

[10] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL implementation secrets. In *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002. 26

[11] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004. 26

[12] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. *SIGCOMM Comput. Commun. Rev.*, 35(4):265–276, 2005. 2, 20

[13] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE software*, pages 34–41, July 1995. 15

[14] S. Bradner. The internet standards process – revision 3, October 1996. 13

[15] A. Brüggemann-Klein. Regular Expressions into Finite Automata. *TCS*, 120(2):197–213, 1993. 42

[16] D. Comer. *Internetworking with TCP/IP*, volume 1, Principles Protocols, and Architecture. 5th edition, 2006. 15

[17] A. David, G. Behrmann, K. G. Larsen, and W. Yi. A tool architecture for the next generation of uppaal. In B. K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2002. 26

[18] E. W. Dijkstra. Notes on structured programming. *Structured Programming, Academic Press*, 1972. 11

[19] C. Djouvas, N. Griffeth, and N. Lynch. Testing self-similar networks. In *MBT 2006*, 2006. 131

[20] M. A. Fecko, P. D. Amer, M. U. Uyar, and A. Y. DualeIn. Test generation in the presence of conflicting timers. In *TestCom*, page 301b•“320, 2000. 19

[21] M. A. Fecko, M. U. Uyar, A. Y. Duale, and P. D. Amer. Efficient test generation for army network protocols with conflicting timers. In *MILCOM*, Los Angeles, CA, 2000. 19

[22] S. J. Garland and N. A. Lynch. The IOA language and toolset: Support for mathematics-based distributed programming. 1998. 15, 22, 40, 156

[23] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ESEC / SIGSOFT FSE*, pages 257–266, 2003. 158

[24] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997. 5, 104

[25] N. D. Griffeth, Y. Cantor, and C. Djouvas. Testing a network by inferring representative state machines from network traces. In *ICSEA*, page 31. IEEE Computer Society, 2006. 29

[26] N. D. Griffeth and C. Djouvas. Experimental method for testing networks. In *Software Engineering Research and Practice*, pages 935–941, Las Vegas, 2005. 29

[27] N. D. Griffeth, R. Hao, D. Lee, and R. K. Sinha. Integrated system interoperability testing with applications to VOIP. In *FORTE*, pages 69–84, 2000. 2, 19

[28] J. Hajek. Automatically verified data transfer protocols. In *4th ICCC*, pages 749–756, Kyoto, 1978. 1

[29] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990. 15

[30] G. Holzmann. *The SPIN model checker*. Addison Wesley, Boston, MA, 2003. 26

[31] IEEE Computer Society. IEEE standard for local and metropolitan area networks media access control (mac) bridges. *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pages 1–269, 2004. 54, 87, 157

[32] IEEE Standards Association. http://standards.ieee.org/. 14

[33] Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html. 20, 84, 157

[34] Institute of Electrical and Electronics Engineers (IEEE). http://www.ieee.org/. 13

[35] D. Kaynar, N. Lynch, and S. Mitra. Specifying and proving timing properties with TIOA tools. In *RTSS*, 2004. 25, 73, 156

[36] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. revised and shortened version of technical MIT-LCS-TR-917a. March, 2005. 15, 25

[37] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers. 42

[38] J. Kurose and K. Ross. *Computer Networking: A Top Down Approach Featuring the Internet*. Addison Wesley, Longman, 2001. 13

[39] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. 25, 73, 156

[40] D. Lee and K. Sabnani. Reverse-engineering of communication protocols. In *ICNP*, pages 208–216, 1993. 18

[41] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines. a survey. *IEEE*, 84:1090–1126, 1996. 2, 15, 17, 20

[42] D. Lee and M. Yannakakis. Pythia: a software tool for testing data portions of network protocols, technical memo, bell-labs, lucent technologies. 1999. 20

[43] M. Leonard, E.; Archer. Extended abstract: organizing automaton specifications to achieve faithful representation. *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pages 245–246, 11-14 July 2005. 73, 78, 157

[44] H. Lim, D. Kaynar, N. Lynch, and S. Mitra. Translating timed I/O automata specifications for theorem proving in PVS. In *Formal Modelling and Analysis of Timed Systems (FORMATS'05)*, Uppsala, Sweden, September 26 - 28, 2005. 77

[45] B. Littlewood and D. Wright. Stopping rules for the operational testing of safety-critical software. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 444, Washington, DC, USA, 1995. IEEE Computer Society. 13

[46] B. Littlewood and D. Wright. Some conservative stopping rules for the operational testing of safety-critical software. *IEEE Trans. Softw. Eng.*, 23(11):673–683, 1997. 13

[47] N. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, San Francisco, CA, 1996. 15, 22, 132, 135, 138

[48] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989. 22

[49] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematics Studies, Princeton University Press*, 34:129–153, 1956. 5, 17, 157

[50] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Hoboken, NJ, 2004. 11, 12, 159

[51] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607of Lecture Notes in Computer Science, pages 748–752, 1992. 27, 77, 157

[52] F. Risso and M. Baldi. Netpdl: an extensible xml-based language for packet header description. *Comput. Netw.*, 50(5):688–706, 2006. 121, 123

[53] J. Robert W. Buchanan. *The art of testing network systems*. John Wiley & Sons, Inc., New York, NY, USA, 1996. 21

[54] Tcpdump Official Site. http://www.tcpdump.org/. 123

[55] The Internet Engineering Task Force. http://www.ietf.org/. 14

[56] The RFC-Editor. http://www.rfc-editor.org/. 14

[57] VeroModo Inc. Tempo, http://www.veromodo.com/tempo/ (February 2008). 25, 73, 156

[58] C. West. General technique for communications protocol validation. *IBM Journal of Research and Development*, 22(4):393, 1978. 1, 14

[59] R. Williams. The triangle classification problem, August 29, 2002. 11

[60] J. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990. 15

[61] Wireshart Official Site. http://www.wireshark.org/. 123