Formal Verification of Time-Triggered Ethernet Protocol using PRISM Model Checker

Marwan Ammar

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical & Computer Engineering) at

Concordia University

Montréal, Québec, Canada

August 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Marwan Ammar

Entitled: Formal Verification of Time-Triggered Ethernet Protocol using PRISM Model Checker

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science (Electrical & Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Dr. Rabin Raut

______________________ Dr. Youmin Zhang

______________________ Dr. Abdelwahab Hamou-Lhadj

______________________ Dr. Otmane Ait-Mohamed

Approved by ______________________

Chair of the ECE Department

__________ 2011 ______________________

Dean of Engineering

ABSTRACT

Formal Verification of Time-Triggered Ethernet Protocol using PRISM Model Checker

Marwan Ammar

The increase in number of computer systems in almost every aspect of our lives dictates how important it is that they perform their functions properly. Traditionally, there are two ways used to check the correctness of a software system: Testing and Formal Verification. Software Testing is the process of executing a program or system with the intent of finding errors. This is done by feeding input to a software system and checking that the output is the expected one, thus checking the correctness of the system. In a parallel way, Formal Verification asserts the behavior of a software system by making use of mathematical logics to prove that the software's model abides to certain precisely expressed functional statements of correctness. The advantage of Formal methods of verification is that they are able to check (and ultimately prove) correctness over all the possible states of a model. As software systems exponentially increase in complexity over a short period of time, their correctness becomes something of utmost importance, especially in safety-critical scenarios, such as aerospace communication and health care. For this reason, Formal Methods are often preferred over testing for both their scope and their proof power. One approach to Formal Verification, which is of interest to this work, is Model Checking, that consists of a systemic exhaustive exploration of all reachable states and transitions of a mathematical model. In this thesis we propose a structure and a conceptual model of a real-time safety-critical communication network based on the fault-tolerant Time-Triggered Ethernet (TTE) protocol for Local Area Network (LAN) environments. TTE is a relatively new LAN protocol that's becoming widely used for its reliability, versatility and bandwidth capabilities, all of which combined make the TTE the best solution for real-time safety-critical scenarios. We derive a set of precisely expressed functional statements (properties), based on the TTE specification document, and we encode our conceptual model into PRISM (Probabilistic Symbolic Model Checker) tool. We use the version 3.3.1 of the PRISM tool to perform the model checking of the network model against the defined properties. Finally, we propose a methodology to generate counterexamples for a property that has failed in PRISM model checking. In the verification of our model we were able to reach one state in the network that fails in two out of our set of eight TTE properties. We apply our methodology, using PRISM's simulation mode, to search through a sample of visited states in order to find and present the counterexample to the end user in a easy and comprehensive way that can, in theory, be applied to any model compliant with our methodology.

ACKNOWLEDGEMENTS

It has been an amazing experience to accomplish my Master's thesis in the Hardware Verification Group (HVG) at Concordia. It certainly would not have happened without the support and guidance of several people to whom I owe a great deal.

First of all, I would like to thank my supervisor, Dr. Otmane Ait Mohamed. It is he who offered me the opportunity to join the group. He was fully supportive, understanding, involved and present during all the phases of my research. I have learned many things from him in regard to research, academia, and life in general.

Secondly, I sincerely thank Prof. Mohamad Sawan, for always being there to listen and to offer me guidance in the moments I needed the most. This thesis would not have been possible without his advice, his support and his encouragements.

Next, I'd like to thank all the members of HVG for their help and encouragement, particularly to Mr. Ghaith Bany Hamad. Their friendship brought an enjoyable work atmosphere in the labs.

I'd like to give especial thanks to my friend Guilherme Schuch for being like a brother for as long as I can even remember.

Last but not least, I thank my close family and my parents. They have a great deal of credit on everything I have accomplished in my life and for everything I'm yet to accomplish in the future.

To my parents, who made every step on my life possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

TTE	Time-Triggered Ethernet
PRISM	Probabilistic Symbolic Model Checker
LAN	Local Area Network
BDD	Binary Decision Diagram
TTP	Time-Triggered Protocol
TDMA	Time-Division Multiple-Access
TT	Time-Triggered
ET	Event-Triggered
PTTM	Protected Time-Triggered Messages
UTM	Uniform Time Format
PTA	Probabilistic Timed Automata
MDP	Markov decision processes
SEE	Single Event Effects
SET	Single Event Transit
CDMA	Code division multiple access
PCTL	Probabilistic Computation Tree Logic
FSM	Finite-State Machine
TP	Theorem Proving
TL	Temporal Logic
LTL	Linear-Time Logic
BDD	Binary Decision Diagrams

Chapter 1

Introduction

In software and hardware systems, Formal Methods are defined as a set of mathematically-based techniques used for specifying what the system ought to do, modeling what the system actually does and verifying if the model respects the specification. These techniques can be used to describe a complete or partial system behavior at various different levels of abstraction which can be used as input to an automated tool. The tool will, then, make use of mathematical proof to ensure correct behavior.

Formal methods assert their system design through the use of formal verification schemes, where the basic principles of the system must be proven correct before they are accepted [25]. Traditional system design has used extensive testing to verify behavior, but testing is capable of only finite conclusions. Testing can only assert that the given system will not fail in the preset test scenarios, which does not mean that the system is bug-free. In contrast, formal methods deal with all possible states of a system so once a system is formally verified, it is guaranteed to be bug free.

Formal design is mainly composed of three steps:

1. Formal Specification - In this step the system has to be specified in a modeling language, which bears a formal, unambiguous grammar (in our work we have used PRISM language).

2. Verification - By describing a system using a formal language, the designer is actually developing a set of theorems about his system. In the verification step the designer tried to validate those theorems in a formal verification tool.

3. Implementation - Once the model has been specified and successfully verified, it is implemented by converting the specification into a real system.

There are mainly two ways of doing formal verification: Theorem Proving and Model Checking.

1.1 Theorem Proving

Theorem Proving (also known as Automated Theorem Proving or Automated Deduction) [26] deals with the development of computer programs that show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). Theorem Proving (TP) can be used to solve problems in a wide variety of domains given an appropriate formulation of the problem as axioms, hypotheses, and conjectures.

The language in which these conjectures, hypotheses and axioms (formulae) are written takes the form of a language that express logics, either classical first order logic or a higher order logic. In these languages, it is possible to precisely state the necessary information, in a formal manner, which can be manipulated by a TP tool. This formality is the biggest strength of TP, and of Formal Methods in general: there is no ambiguity in the statement of the problem, as is often the case when using a natural language.

The results produced by a TP tool come in the form of proofs that validate the described system, showing that the conjecture is a logical consequence of the axioms and hypotheses.

TP tools are very powerful and complex computer programs, capable of solving very difficult problems. As a rresult, their application and operation often needs to be guided by an expert, in order to solve problems in a reasonable amount of time. Thus

TP tools are often of interactive nature, where the program solves parts of the problem and asks for user input whenever it reaches an impasse. The interaction may be at detailed level (where the user guides the inferences made by the system) or at a higher level (where the user determines intermediate lemmas to be proved on the way to the proof of a conjecture). The steps in using a TP tool can be roughly represented as follows:

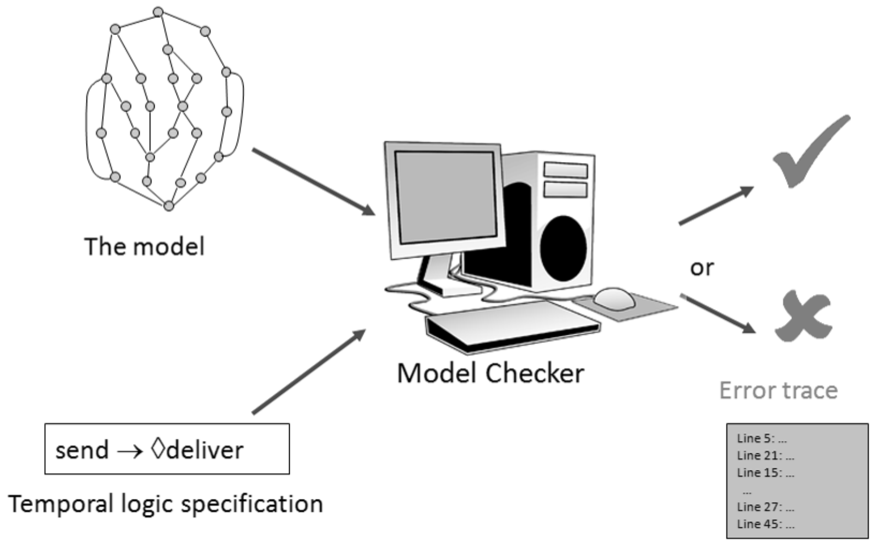
- The tool needs a precise description of the problem written in logical form.
- The user has to think carefully about the problem in order to produce an appropriate formulation and hence acquires a deeper understanding of the problem.
- The tool attempts to solve the problem.
- If successful the proof is given as output.
- If unsuccessful the user can provide guidance or try to break down the problem into smaller problems and try to prove them, or examine the formulae to make sure that the problem is correctly described.
- Iterate the previous step until a proof is obtained.

TP is, thus, a technology suited for big and complex verification scenarios where an automatic result is not expected but where an expert will interact with a powerful tool in order to solve the problem. There are many TP tools readily available for use. Some of the tools that deal with First Order Logics are: Otter, E, SPASS, Vampire, and Waldmeister. Tools for Higher Order Logics include: ACL2, Coq and HOL.

1.2 Model Checking

Model checking is a Formal Method that designates a collection of techniques for the automatic analysis of reactive, finite state concurrent systems, such as sequential circuit designs and communication protocols. This technique has several advantages over simulation, testing, and deductive reasoning. Notably, Model Checking is automatic and usually very fast. This means that given a model of a system, Model Checking will test automatically whether or not this model meets a given specification. If the design of the system contains any errors, Model Checking will identify them and will produce a counterexample that can be used to pinpoint the source of the error.

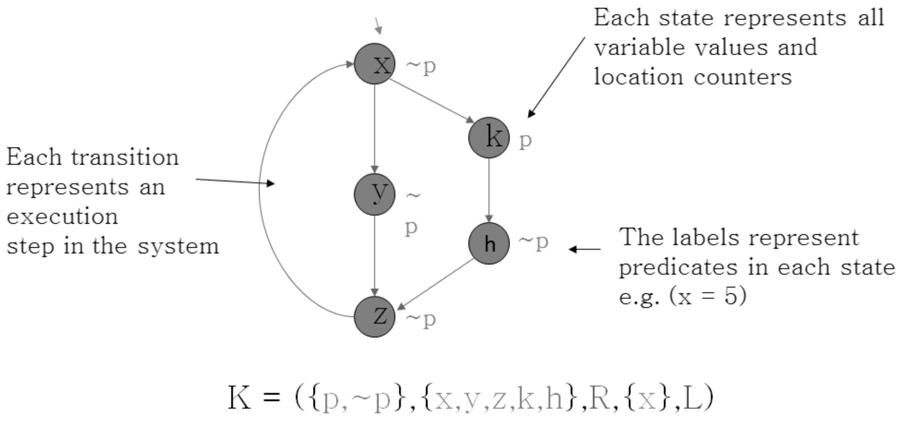
The main idea behind Model Checking, like in Theorem Proving, is to formulate the system and the specification in a precise mathematical-based language and feed them to a verification tool that will check if the given system structure satisfies the specification (Figure 1.1).



```
Line 5: ...
Line 21: ...
Line 15: ...
  ...
Line 27: ...
Line 45: ...
```

Figure 1.1: Verification via Model Checker

The input to a model checker is usually a description of the system to be analyzed, in the form of a finite-state machine (also known as Kripke Structures, Figure 1.2) [39], and a set of properties that express the desired functionality of the system, commonly expressed in temporal logic formulas. The model checker, then, will either confirm that the properties hold true within the model or it will report which properties are violated and often the model checker is also capable of generating a counterexample that shows the path in the model that leads to the property violation.



Each state represents all variable values and location counters
Each transition represents an execution step in the system
The labels represent predicates in each state e.g. (x = 5)
x
~p
k
p
y
~p
h
~p
z
~p
K = ({p,~p},{x,y,z,k,h},R,{x},L)

Figure 1.2: Example of a Kripke Structure

Properties in model checking are written in Temporal Logic (TL). TL is a form of symbolic reasoning to express the ordering of events over time. TL formulas serve as precise, concise and binding descriptions of systems and components [30, 31]. TL can be of two types:

Linear Time: Linear-time logic (LTL) considers behaviors modeled as linear sequences of states in function of time. Within one behavior chain, each state has only one path ahead of it.

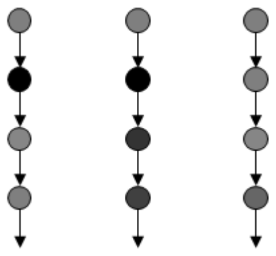


Figure 1.3: LTL pattern, [39]

Branching Time: In Branching-time logic, the formulas refer to behaviors structured like trees, where each state has several possible paths ahead of it. These behavior-trees are ideal to describe the models of non-deterministic systems. A prominent type of branching-time logic is the Computation-Tree Logic (CTL) [32].

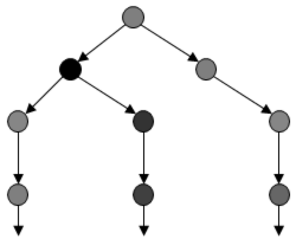


Figure 1.4: CTL pattern, [39]

The main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. Many techniques are employed to handle this problem (notably Partial Order Reduction [28]) by reducing the total number of reachable states of a system by eliminating equivalent paths or analyzing only paths that are relevant to the properties. These techniques make model checking possible for bigger designs but they sacrifice the accuracy of the verification as a faulty state could be hidden in one of the many eliminated paths. Thus model checking can prove that a system design contain a failure but cannot prove that the system does not contain any failure [29].

Probabilistic Model Checking

Probabilistic model checking is a formal verification technique, derived from regular model checking, applied on systems that present a random or probabilistic behavior, like real life applications, where the resulting models usually contain a very large number of states. It would be very impractical for the user to explicitly model every state and transition of these applications. The solution is to provide a high-level specification formalism to describe real life systems in a meaningful and simple way.

One of the most notable features of this technique is not only being able to receive probabilities as an input but also to return probabilities as output. Given that a perfect, bug-free system is something near impossible to achieve, probabilistic model checking allows the user to work with tolerance percentages rather than absolute values -something that's very common in communication protocols.

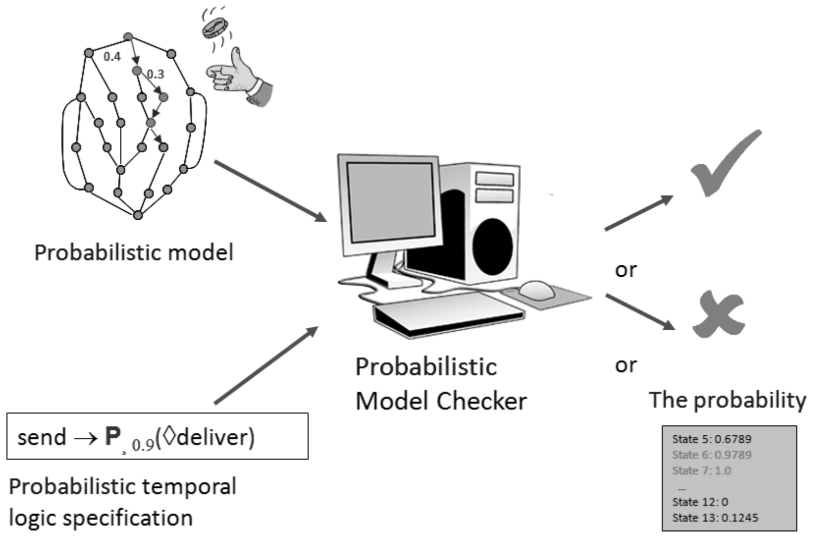


Figure 1.5: Verification via Probabilistic Model Checker

Probabilistic model checking can deal with a wide range of quantitative measures, the results show an exact figure of the property being verified (usually in parts-per-hundred), it can be fully automated (model construction plus numerical solution), provides an exhaustive analysis of the model (accounting all possible initial states and model parameter values as well as all possible process scheduling) and it is very efficient. The downside is that even though it can identify patterns, trends and anomalies in quantitative results, most tools cannot provide counterexamples and, like regular model checking, it suffers from state-space explosion.

Probabilistic model checking works with several model types and temporal logic specification languages. For our work we use Markov Decision Process (MDP) as the model type and Probabilistic Computation-Tree Logic (PCTL) as the property specification language.

1.3 Thesis Contributions and Related Work

Given that Time-Triggered Ethernet (TTE) is a relatively new protocol, the amount of research work done on top of it is very limited and the majority of the works come from the same research group. In most of the previous the focus is mainly on design and implementation of the protocol or part of it. We list the most relevant related works done using TTE protocol.

In [6] the research team develops a software implementation of a TTE controller for real-time applications, built on top of a 100Mbit Ethernet controller. Their work covers important topics about TTE, like its architecture and functionality and it presents the controller's structure for TTE, giving results in form of performance analysis and evaluation of test cases.

In [7] the research team works on the hardware-level architecture of a TTE switch, presenting how it handles Time and Event-triggered messages and discussing its general behavior at low-level and how the switch handles predictable real-time communication as well as being able to work with other networking protocols, such as regular Ethernet. The work's main focus lies on the TTE switch architecture and its functionality. The authors gave experiment results comparing the TTE switch with COTS Ethernet switch.

In [2] we have a very comprehensive presentation of the elements that are present in a TTE setting including all of its main functions and services, its basic definitions, the protocol's requirements, different settings and message types and formats. In our own research, while designing our scenarios, we used this work as a second TTE specification, as it is presented in a very concise and easy to read format.

In [3] the authors discuss an integration of TTE with SpaceWire networks for aerospace applications. This work covers the dataflow and synchronization in TTE and presents the benefits of an integrated TTE-SpaceWire architecture, giving four integration options and the main ideas behind each of them.

The work in [4] is the only related work we could find that uses a formal method to verify a part of TTE protocol. This work focus on the Compression Function of TTE, which is a clock synchronization function that runs inside the TTE switches collecting clock data from the connected systems and calculating a synchronized figure to send to the connected systems as feedback. The formal verification technique used in that work is Yices SMT Solver [8].

In the work presented by this thesis, the focus lies in two related objectives. The first objective is to model a state-driven closed-world scenario encompassing all the elements of a network, making use of the TTE standard, and then formally verify it using PRISM Model Checker. Our design is largely based on the descriptions available in [2, 3, 6, 7, 15] and we take [4] into account for modeling our clock synchronization function. To our knowledge, this is the first time TTE protocol has been verified as a complete network scenario at high-level abstraction, which allows us to study the behavior of the TTE network and verify all the relations between its multiple components and through our verification we were able to identify the existence of a failing state in TTE protocol.

The second objective is to propose a methodology for generating a counter-example for the properties that did not pass the formal verification, since PRISM does not do that automatically. This methodology involves an specific way of coding the model into PRISM language and the implementation of a software that makes use of a table of states extracted from PRISM to search and find for the state that violates the property.

1.4 Thesis Outline

The rest of this thesis is organized as following:

- In chapter 2 we present the Time-Triggered Ethernet Protocol with its main properties and functions.
- In chapter 3 we give an introduction to PRISM Model Checker, to Markov Decision Processes and to Probabilistic Computation-Tree Logic.
- In Chapter 4 we present our modeling of two TTE network scenarios, we describe the functionality of each one of our modules and we discuss the assumptions involved in the process.
- In Chapter 5 we present and explain the set of properties that we are verifying in the TTE scenarios, we propose a methodology for generating a counter-example based on PRISM verification and simulation, we discuss the results of the formal verification on both network scenarios and we identify the path that leads to a integrity failure in the TTE protocol.
- In Chapter 6 we conclude this thesis by summarizing our work and discussing some directions that future researches might take.

Chapter 2

Time-Triggered Ethernet

Real-time computer systems have been growing fast in use and complexity over the past decade. As their complexity grows, reliability becomes a major concern, especially for safety-critical systems such as aerospace and health care applications. Fault-tolerant communication protocols have been created and are increasingly being used to ensure reliable end-to-end communication with minimum data loss even in case of catastrophic hardware failure on these real-time safety-critical environments. The problem that arises is how to certify that a fault-tolerant protocol built for a safety-critical scenario is in fact free of faults for every possible state and transition. Traditionally, network protocols (even very important ones such as the Ethernet protocol) have been proven by exhaustive testing and debugging faults inherent to their designs over time (what's called field proving). For obvious reasons, field-proving is not a viable alternative when the application holds vital real-time information and thus fault-tolerant protocols are conceived, at least in part, using formal methods. In this research we make use of the Time-Triggered Ethernet (TTE [1], [2], [15]), which is a new Local Area Network protocol conceived with the objective of creating a universal real-time and fault-tolerant solution to be used on safety-critical communication systems. TTE builds on top of Ethernet and adds to it a deterministic time and event triggered communication layer. The infrastructure of this novel protocol is designed for mixed criticality systems, allowing transit of data from different dataflows from applications bearing different criticality levels such as aerospace and health-care applications, data-backup networks or regular LAN communication. Despite its high level of sophistication and the use of formal methods in the development of some of its fault-tolerant algorithms, the TTE protocol has never been verified as a whole functional network scenario and, as stated in the specification document, it may contain faults. Thus, implementation for safety-critical environments, as of today, still relies heavily on testing a representative set of states, in search for possible fault occurrences, and physical redundancy to ensure fault-free functionality. Our work aims to address the aforementioned issue by modeling a state-driven closed-world scenario encompassing all the elements of a network, making use of the TTE standard, and then formally verify it using PRISM (Probabilistic Symbolic) Model Checker. On a second moment, our work aims to address a limitation of PRISM tool given that by the time this work was done the tool still was not able to generate a counter-example for a failing state.

2.1 TTE Description

Time-Triggered Ethernet is a communication system that extends upon the functionality of regular Ethernet protocol by merging Ethernet with Time-Triggered Protocol (TTP), adding a deterministic time-driven layer to Ethernet communication. TTE supports distributed non-real-time and real-time applications to work simultaneously on the same network environment, allowing existing Ethernet applications to be ported to TTE format without any hardware or software modification.

The principle on which TTE bases its functionality upon is the creation a time partition among all the nodes within a network, assigning a time frame for each node in which that node will be allowed to send messages. This is known as Time-triggered communication or time-division multiple-access (TDMA) and it is achieved by synchronizing the clocks of all the participants of a network, allowing a deterministic system behavior where the possibility of data collision due to two nodes transmitting at the same time is excluded by design and enforced by guardian functions when applicable.

TTE has a well defined hierarchical structure, centered upon the switch (single or multiple) which is the heart of the network, where all main control functions are located, implemented in either hardware or software level, depending on the case. The switches are responsible for maintaining the following properties in the network at all times, for they are requirements of TTE:

1. Fault-tolerant Global Time - after the initial synchronization step, all entities should operate according to the global clock at all times (see synchronization function below).

2. Seamless Communication Architecture - dictates that a TTE network should be able to handle mixed-networks of mixed-criticality, composed by either Time-Triggered (TT) and Event-Triggered (ET) without any additional configuration.

3. Deterministic Communication - all communication should follow a predictable pattern at all times.

4. Strong Fault Isolation - any fault that happens in the network, either on hardware or on logical level, has to be contained, according to the TTE Fault Hypothesis (see Fault Hypothesis below).

5. No Single Point of Failure - TTE should be able to handle any arbitrary point of failure without compromising functionality.

6. Naming - all messages types in TTE should be identifiable by name.

7. Consistent Diagnosis - any fault should be traceable.

The TTE switch is also responsible for the synchronization and guardian functions, which are detailed in their own subsections below. TTE can support either Time-Triggered or standard Event-Triggered messages, but on this work we only care for Time-Triggered messages.

Formally, we define TTE protocol in the Definition 2.1.1.

Definition 2.1.1 (TT Ethernet Protocol)**.** *A TT Ethernet Protocol is a tuple* $TT = (\mathbf{Obj}, \mathbf{Int}, \mathbf{Order})$ *where:*

- $\mathbf{Obj}$ *is the set of objects that define the actors such as server, clients and switchers,*
- $\mathbf{Int} = m_1, \ldots, m_n$ *is a sequence of messages of size* n*. Each message has the form* $m_i = (\mathbf{S_i}, \mathbf{N_i}, \mathbf{G_i}, \mathbf{T_i})$ *where* $\mathbf{S_i}$ *is the source,*$\mathbf{N_i}$ *is the message content,*$\mathbf{G_i}$ *is a guard and* $\mathbf{T_i}$ *is the message target*
- $\mathbf{Order} : \mathbf{m_i} \to \mathbb{N} \times \mathbb{N}$ *is a function assigning to each message its order in the source and the target objects.*

2.2 Synchronization Function

The clock synchronization function in TTE is the service that's responsible for providing the initial synchronization and for ensuring that all local clocks in the communication system stay synchronized. The way it handles synchronization to global time varies according with the setting. It assumes a Master/Slave position in Standard TTE Configuration and a Multi-Master stance in Fault-tolerant TTE configuration.

The two possible scenarios in the Synchronization step, according to the number of Synchronization Masters are:

1. Master/Slave One-Step Synchronization - this is the case where there is only one Synchronization Master. Here the single Synchronization Master is also the Compression Master of the network and the Compression Function will be executed in one switch only, resulting on a global latency that matches the switch's local latency.

2. Multi-Master Two-Step Synchronization - in this case we have more than one switch in the network. All the switches are Synchronization Masters and one of them is the Compression Master.

Global time synchronization is achieved through a service called Compression Function. The Compression Function is executed to collect and compress the Control Frames, dispatched by the Synchronization Masters (multiple Synchronization Masters in case of a Multi-Master network or a single Synchronization Master in case of a Master-Slave network), into one Compression Master. In other words, the Compression Function calculates the network latency, based on each of the Synchronization Master's local latency, in order to make all Synchronization Masters to work in Synchrony.

Figure 2.1 [1] offers an overview of how the Compression Function operates.

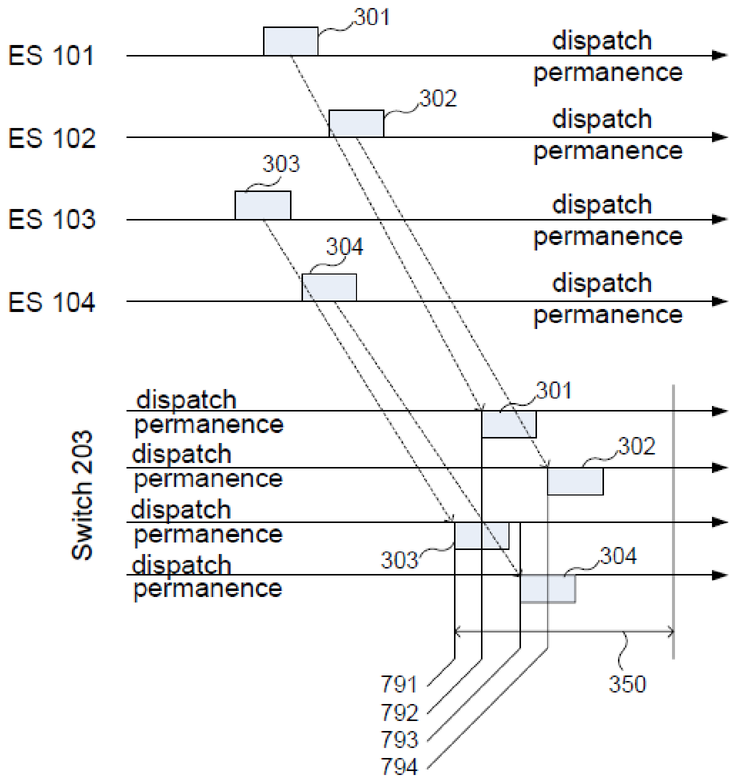


Figure 2.1: Overview of the Compression Function

After the network masters are in synchrony, the Synchronization Function enters the Integration Cycle step, where the global clock will be transmitted to the rest of the network. Figure 2.2 [1] contains an overview of the Integration Cycle.

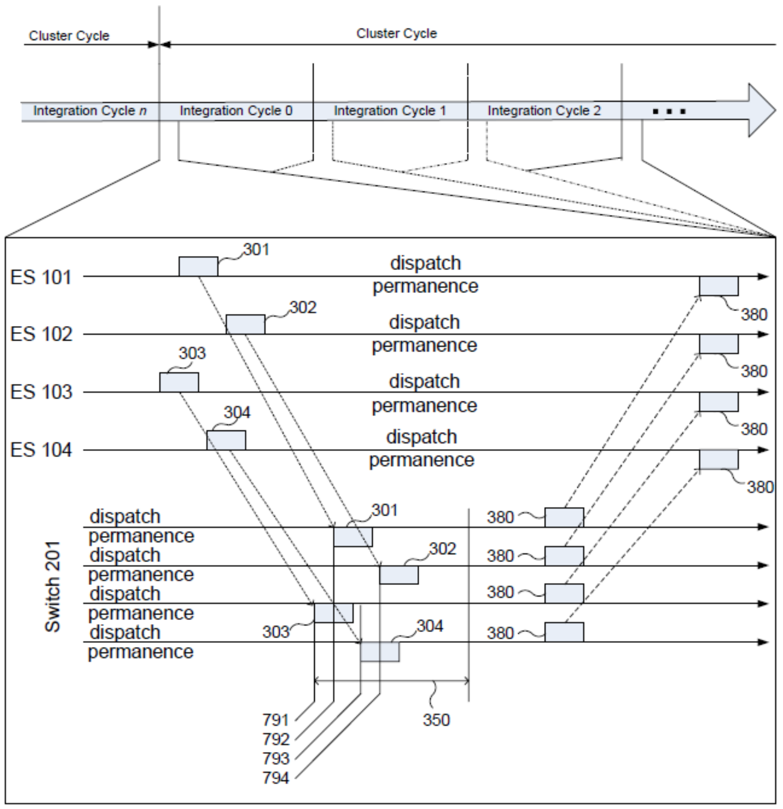


Figure 2.2: Example of Integration Cycles

The procedure is similar to what happens in the Compression Function, except that the Compression Function works with latencies among Synchronization Masters and the Integration Cycle works with the latencies between Synchronization Masters and Synchronization Clients. At the starting of the cycle in Figure 2.2, the Synchronization Clients have received a Protocol Control Frame dispatched by the Compression Master and the Compression Master have received those Control Frames back. Frames coming from each Client will have a different transmission latency when they arrive at the switch. The compression function, located inside of the Compression Master switch, calculates a time delay to fit the sending nodes into the previously defined global time and sends it to all nodes in the form of new Control Frames. Synchronization cycles occur at regular time intervals allowing new nodes to join the network on-the-fly.

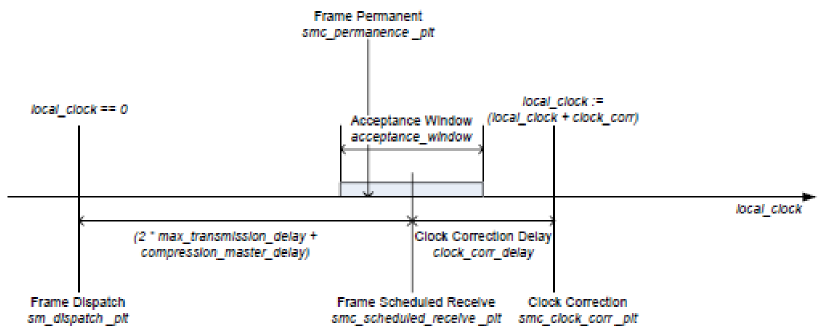


Figure 2.3: Local Clock in Synchronization Function

Figure 2.3 [1] summarizes in a very comprehensive manner how the local clock is calculated inside each of the network's entities, once the Compression Master's delay have been established.

2.3 Fault Hypothesis

The Fault Hypothesis in TTE consists of a series of measures meant to ensure fault containment (limiting the impact of a single fault to a defined region) and error containment (avoiding the propagation of the consequences of a fault). TTE can be divided into two categories based on how the network is built to deal with fault and error containment: Standard TTE and Fault-Tolerant TTE.

2.3.1 Standard Time-Triggered Ethernet

Standard TTE configuration consists of a set of computer nodes connected to a TTE switch. A computer node is formed by a host computer system and a standard communication controller. A generic star network topology for this configuration is shown in Figure 2.4 [15].

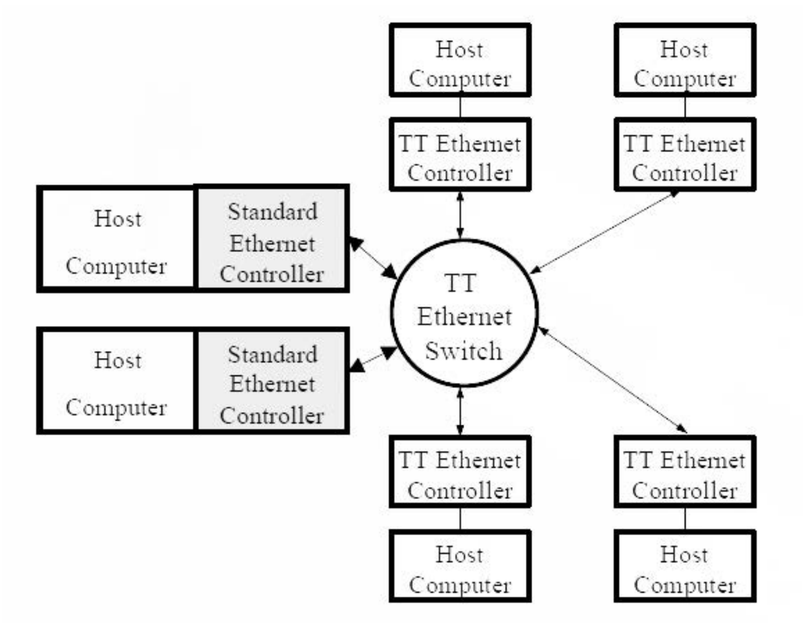


Figure 2.4: Standard TTE Configuration

This configuration is ideal for real-time applications that broadcast a constant stream of data during a period of time and require a constant transmission delay to maximize efficiency, such as multimedia and data-backup applications.

Standard TTE uses a simple Master/Slave one-step synchronization function, where the single master (switch) will calculate the synchronization delay, based on its internal clock, and broadcast it to the rest of the network. This clock will be adopted as the global time reference in all the nodes connected to this network.

Standard configuration is built around the single-point Failure Hypothesis, which means that the network can tolerate any kind of hardware failure without compromising its integrity, as long as there is only one point of failure at any given time. This means that the network's functionality should not be compromised by any kind of fault or error that may be generated during communication. Any failure or error should be identified and contained as fast as possible.

2.3.2 Fault-tolerant Time-Triggered Ethernet

Fault-Tolerant TTE Configuration is used for safety-critical applications and it consists of a set of computer nodes connected to two or more TTE switches. A generic star network topology for this configuration is shown in Figure 2.5 [15].

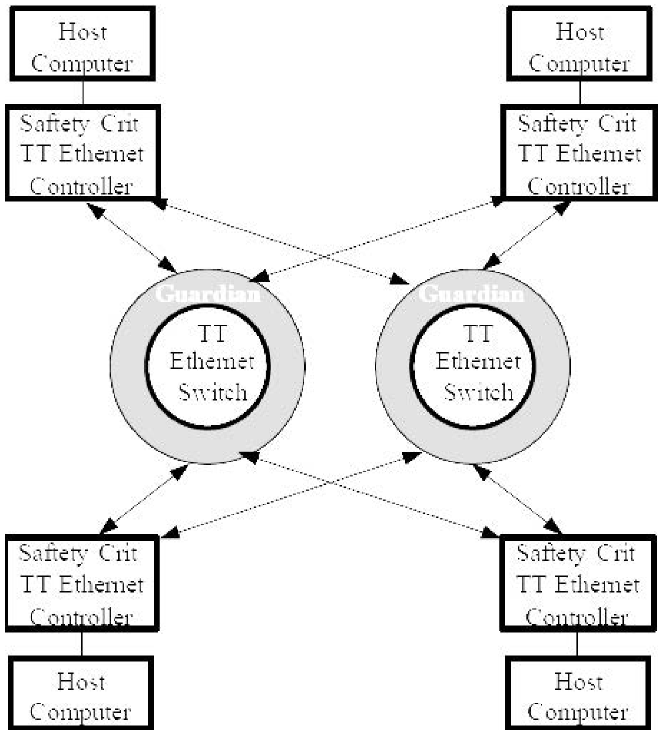

Host Computer
Saftety Crit TT Ethernet Controller
Host Computer
Saftety Crit TT Ethernet Controller
Guardian
TT Ethernet Switch
Guardian
TT Ethernet Switch
Saftety Crit TT Ethernet Controller
Saftety Crit TT Ethernet Controller
Host Computer
Host Computer

Figure 2.5: Fault-Tolerant TTE Configuration

As it can be seen in the figure above, Fault-tolerant TTE have an additional service, compared to Standard TTE, called Guardian Function, the nodes feature especial safety-critical controllers and all the connections are redundant.

1. Safety-Critical TTE Controller - as a safety measure in Fault-Tolerant TTE, all connections are redundant, which means that all messages are also redundant (messages transmitted over two channels are denoted Protected TT Messages -PTTM- because these messages are protected and controlled by the Guardians). This allows the network to tolerate channel failures.

2. Guardian Function - Switches in both Standard and Fault-tolerant configurations are very similar. The main difference is that the input and output ports of the switch in Fault-tolerant configuration are monitored by the switch's guardian. The guardian is a piece of hardware, with its own fault-tolerant clock synchronization sub-system, connected to the switch via ethernet port which is able to control the input and the output of the switch. The Guardians act as a flow control system, able to alter the delay of outgoing messages or block messages from entering the switch. The Guardian can be implemented with a Semantic Filter function -whenever a device receives a control frame in port X it checks if that control frame is allowed to be received on that port- or with a Leaky Bucket function (Figure 2.6 [1]) -the switch limits the number of frames received from a node for a configurable amount of time.

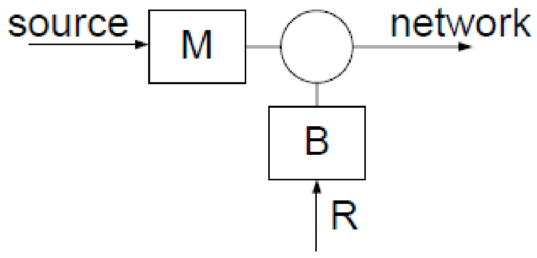


Figure 2.6: Leaky-Bucket diagram

3. Global Time - The global time in TTE is based on the Uniform Time Format (UTF). This time-format has been standardized by the OMG in the small transducer interface standard [16]. A digital time format can be characterized by two parameters: granularity (interval between two adjacent ticks of a digital clock) and horizon (instant when the time will wrap around). TTE's time format is a 64-bit binary time-format that is based on the physical second (defined internationally in terms of radiation emitted by caesium atoms). Fractions of a second are represented as 24 negative powers of two (the granularity is approx. 60 nanoseconds), and full seconds are presented as 40 positive powers of two (the horizon is about 30,000 years), as represented in Figure 2.7 [1].

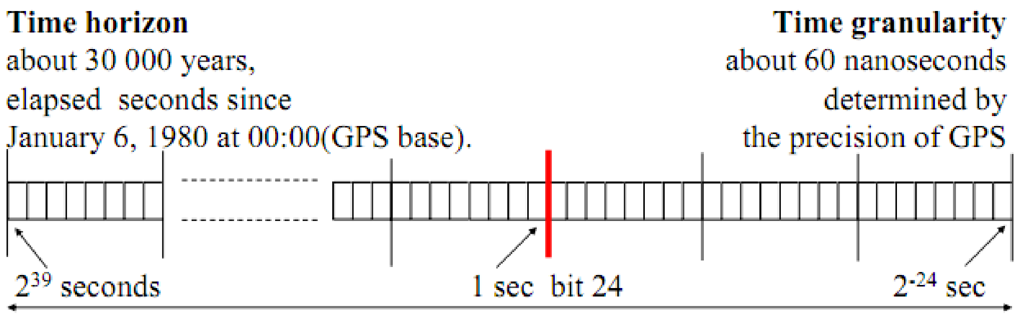


Figure 2.7: TTE Time Format

Chapter 3

PRISM Model Checker

3.1 PRISM Overview

PRISM [10] is a free, open source probabilistic symbolic model checker developed at the University of Birmingham. It works with its own high-level modeling language, based on the Reactive Modules formalism [17], which is written in form of state-based modules, each composed by a set of guarded commands. PRISM uses Binary Decision Diagrams (BDD) (Binary Decision Diagrams) and Multi-Terminal Binary Decision Diagrams (MTBDD) [33] to construct and compute the reachable states of even very large probabilistic models.

PRISM is a very flexible tool to work with probabilistic real-life models as it allows for the specification of probabilities inside the model and in the properties. Additionally the software will inform what's the probability of given property failing after the verification. PRISM allows for step-by-step simulation where the user may chose which variables on the system he wants to manipulate as well as their initial values. The simulation may be guided, where the user manually selects the next step to be taken, or random, where the user selects the number of random steps the program should simulate.

PRISM also offers great ease in running experiments on a model. The user may select one of the system's properties and run an experiment simulation with it, where he may define the range of values that the variables related to that property should assume and also the rate their values are incremented during this experiment. Prism will verify the selected property considering the values entered by the user and it will plot a graph with the probability of the property to be true at every one of the variable's multiple values.

In PRISM the user defines his model through processes, each composed by several guarded actions. The execution of the commands follows the logic: if guard is true, then execute the actions (which are variable updates within the model scope).

Another very important feature of PRISM is the ability to do synchronization between modules by making use of action labels. PRISM supports a wide range of model analysis methods and it features a very efficient implementation, making use of a symmetry reduction technique [12] in order to avoid state-space explosion. PRISM works by creating a probabilistic model of the system and computing its reachable states. The model checking is done by dynamically creating graph-based computations [13] in order to reach a numerical solution (based on linear equation systems and optimization problems).

PRISM supports various different types of probabilistic model representation and temporal logic specification languages:

Model Representation:

- Discrete-time Markov Chains.
- Continuous-time Markov Chains.
- Markov Decision Processes.
- Probabilistic Timed Automata.

Temporal Logic Specification:

- Continuous Stochastic Logic - used with CTMC.
- Probabilistic Computation Tree Logic - used with DTMC and MDP
- Probabilistic Timed Computation Tree Logic - used with PTA.

In our work we use Markov Decision Processes as our model and Probabilistic Computation Tree Logic as our temporal logic specification language.

3.2 Probabilistic Verification

Probabilistic model checkers such as PRISM are mainly based on the stochastic version of the classical shortest path problem. This problem was formulated by Eaton and Zadeh [36] who called it the problem of *pursuit*.

In this section, we introduce probability computation in a symbolic model checker. It proceeds by induction on the parse tree of the formula, as in the case of CTL model checking. To show that, we select the MDP as a special [37] formalism of probabilistic automata that exhibit both probabilistic and nondeterministic behavior. It is defined in the Definition 3.2.1.

Definition 3.2.1 (Markov Decision Process)**.** *A Markov decision process (MDP) is a tuple* $M = (\mathrm{S}, \overline{s}, \alpha_{\mathrm{M}}, \delta_{\mathrm{M}}, \mathrm{L})$ *where:*

- S *is a finite set of states,*
- $\overline{s}$ *is an initial state,*
- α_{M} *is a finite alphabet,*
- $\delta_{\mathrm{M}} : \mathrm{S} \times \alpha_{\mathrm{M}} \to \ Dist\,(S)$ *is a (partial) probabilistic transition function,*
- $\mathrm{L} : \mathrm{S} \to 2^{AP}$ *is a labelling function mapping each state to a set of atomic propositions taken from a set AP.*

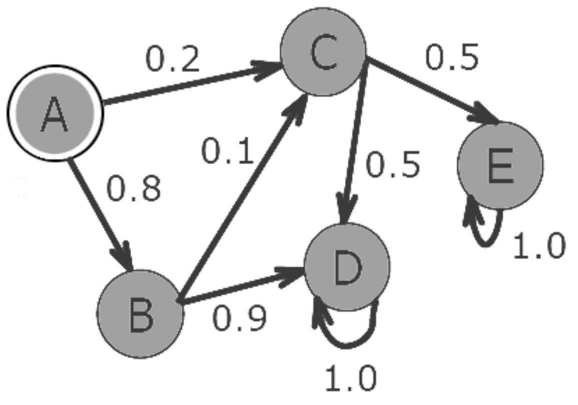


Figure 3.1: Example of a MDP FSM, [40]

To reason formally about MDPs, we need a probabilistic space over them. As MDP follows a nondeterministic behavior, the *adversary* notion is introduced to decide which action should be chosen in any state of the MDP. In general, the choice is made depending on the execution of the MDP. The Definition 3.2.2 describes the adversary function.

Definition 3.2.2 (Adversary)**.** *An adversary of an MDP* $M = (\mathrm{S}, \overline{s}, \alpha_{\mathrm{M}}, \delta_{\mathrm{M}}, \mathrm{L})$ *is a function* $\sigma : FPath_M \rightarrow Dist(\alpha_M)$ *that maps every finite path of the system onto a distribution where:*

- $\sigma(\rho)(a) > 0$ *only if* $a \in A(last(\rho))$*,*
- $FPath_M$ *is a finite set of nodes (states),*
- $Dist(\alpha_M)$ *is a labeled function assigning to each node of the automaton the set of atomic propositions that are true in that node.*

Reachability lies in the core of a model checker. Probabilistic reachability refers to the minimum/maximum probability with which a given set of states of a probabilistic system ($T \subseteq S$) can be reached from a particular state (s). For this, $reach_s(T)$ is the set of paths that start from s and contain a state from T.

$reach_s\,(T) = \{\pi \in IPath_{M,s} | \pi\,(i) \in T and i \in \mathbb{N}\}$
$= \bigcup_{\rho \in I} \{\pi \in IPath_{M,s} | \pi \, has\, prefix\, \rho\}$, where I is the set of all finite paths from s to T, and each element in this union is measurable. This is equivalent to computing the probabilistic bounds of the reached paths:

$$P_{M,s}^{min}(reach_s\,(T)) = inf_{\sigma \in Adv_M} Prob_{M,s}^{\sigma}(s, \psi) \tag{3.1}$$
$$P_{M,s}^{max}(reach_s\,(T)) = sup_{\sigma \in Adv_M} Prob_{M,s}^{\sigma}(s, \psi) \tag{3.2}$$

Bertsekas and Tsitsiklis [36], prove that this minimum probability is the unique solution for Bellman's equation and the successive approximation methods converge to the optimal vector. This yields to the fact that the linear programming problem can be solved as an equation system problem. This means, finding the probability
$x_s = P_{M,s}^{min}(reach_s\,(T)),\ s \in\ S$ is the unique solution of Bellman's equation:

$$\mathrm{X_s} = \begin{cases} 1 & \text{if } \mathrm{s} \in \mathrm{S_{min}^{s=1}} \\ 0 & \text{if } \mathrm{s} \in \mathrm{S_{min}^{s=0}} \\ \min_{\mathrm{a} \in\ \mathrm{A(s)}} \sum \delta_M(s,a)(\mathrm{s}') \cdot \mathrm{X_{s'}} & \text{otherwise} \end{cases} \tag{3.3}$$

The reachability probabilities can be achieved through three methods: value iteration, linear programming problem and policy iteration. Value iteration is the most used technique, due to its approximation algorithm based on an iterative solution method, which corresponds to a fixed point computation.

To complete the model checking process, a property should be specified to verify if it holds true and what's the probability that it will hold true. For that goal, different mechanism are used, such as temporal logic and special automata. In our work, we used a probabilistic extension of CTL temporal logic called PCTL. PCTL is supported by most tools and its BNF grammar is expressed as follows:

$$\phi ::= true|a \mid \phi \wedge \phi \mid \neg\phi \mid \mathrm{P}_{\bowtie p}[\psi]$$

$$\psi ::= \mathrm{X}\phi|\phi\mathrm{U}^{\leq k}\phi|\phi\mathrm{U}\phi$$

where a is an atomic proposition, $k \in N$,$p \in [0,1]$, and $\bowtie\in \{<, \leq, >, \geq\}$. To specify a satisfaction relation of a PCTL formula in a state s, a class of adversaries (Adv) is defined [37]. It is true iff it is satisfied under all adversaries of a given MDP.

The satisfaction relation ($\models_{Adv}$) of PCTL is defined [37] inductively as follows:

$s \models_{Adv} True \quad Always$

$s \models_{Adv} a \Leftrightarrow a \in L(s)$

$s \models_{Adv} \phi_1 \wedge \phi_2 \Leftrightarrow s \models_{Adv} \phi_1 \wedge s \models_{Adv} \phi_2$

$s \models_{Adv} \neg\phi \Leftrightarrow s \not\models_{Adv} \phi$

$s \models_{Adv} \mathrm{P}_{\bowtie p}[\psi] \Leftrightarrow$ $\forall\pi$ an infinite path in M, we have:

- $\pi \models_{Adv} X\phi \Leftrightarrow \pi(1) \models_{Adv} X\phi$
- $\pi \models_{Adv} \phi_1 \, U^{\leq k} \, \phi_2 \Leftrightarrow \exists \, i \geq k.(\pi(i) \models_{Adv} \phi_2 \wedge \pi(j) \models_{Adv} \phi_1 \, \forall \, j < i)$
- $\pi \models_{Adv} \phi_1 \, U\phi_2 \Leftrightarrow \exists \, k \geq 0. \, \pi \models_{Adv} \phi_1 \, U^{\leq k} \, \phi_2$

From the basic PCTL syntax, several other useful operators can derived with a logical equivalences, such as:

1. Future: $F\phi \equiv true \, U \, \phi$
   and $F^{\leq k}\phi \equiv true \, U^{\leq k} \, \phi$.
2. Generally: $G\phi \equiv \neg(F\neg\phi)$
   and $G^{\leq k}\phi \equiv \neg(F^{\leq k}\neg\phi)$.

Here , we will consider the basic PCTL operators “Next and “Until to compute minimum of probability to reach states that satisfy a formula ψ of type $X\phi$ and $\phi_1 U^{\leq k}\phi_2$.

In the case of $\psi = X\phi$, we have:

$P_S^{min}(X\phi) = \min_{\mathrm{a}\in\ \mathrm{A(s)}} \sum_{s' \in\ Sat(\phi)} \delta_M(s,a)(\mathrm{s}') \cdot \mathrm{s}'$

In the case of $\psi = \phi_1 U^{\leq\ k} \phi_2$, we have:

$$\mathrm{x}_\mathrm{s}^\mathrm{l} = \begin{cases} 1 \text{ if s} \in\ \mathrm{Sat}(\phi_2) \\ 0 \text{ if s} \notin (\mathrm{Sat}(\phi_1) \cup\ \mathrm{Sat}(\phi_2)) \\ 0\text{if s} \in\ \mathrm{Sat}(\phi_1) \backslash\ \mathrm{Sat}(\phi_2)\ and\ l = 0 \\ \min_{\mathrm{a}\in\ \mathrm{A(s)}} \sum_{s' \in\ S} \delta_M(s,a)(\mathrm{s}') \cdot \mathrm{x}_{\mathrm{s}'}^{\mathrm{l}-1} \text{otherwise} \end{cases} \tag{3.4}$$

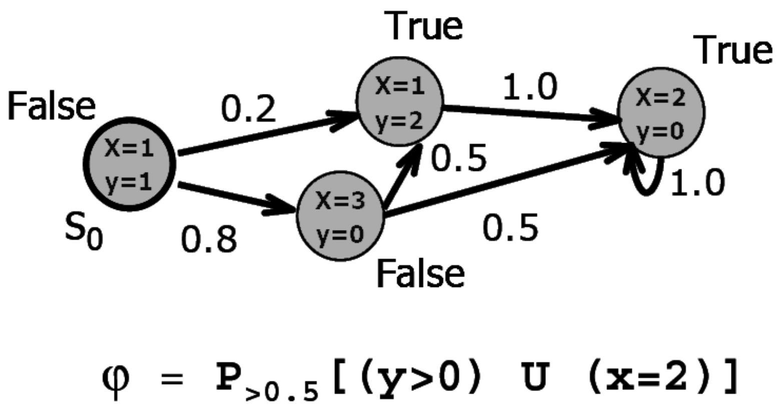


Figure 3.2: Probabilistic State Formula

3.3 Prism Semantic

In this section, we define the PRISM model and its semantic. A system described as a PRISM model comprises a set of n modules. The state of each module is defined by a evaluation of a set of finite-ranging local variables. The global state of the system is the evaluation of the union of all local variables V_l in addition to the global ones V_g, which are denoted $V = V_g \cup\ V_l$. The behavior of each module is defined by a set of guarded commands and a set of invariants in the case of probabilistic timed automata (PTA).

In MDP as in PA and PTA formalisms, a command takes the following form: [act] guard $\rightarrow p_1 : u_1$+...+$p_m : u_m$, and its formal definition is given in the Definition 3.3.1.

Definition 3.3.1 (PRISM Command)**.** *A PRISM command is a tuple* $cmd = (act, guard, update)$ *where:*

- *act: is an action label,*
- *guard: is a predicate over* V*,*
- $update = \{(p_i, u_i)\}$ *is a set of* m *variable updates such that:* $\sum_{i=1}^{m} p_i = 1$ *and* $u_i = (v_i' = val)$ *where* $val \in [v_i^{min}, v_i^{max}]$.

A module that describes the behavior of a sub-part from a system is defined formally in the Definition 3.3.2.

Definition 3.3.2 (PRISM Module)**.** *A PRISM model is a tuple* $M = (name_M, var(M), \overline{var(M)}, com(M))$ *where:*

- $name_M$ *is the name of module M,*
- $var(M)$ *is a finite set of module local variables,*
- $\overline{var(M)}$ *is the initial value of* $var(M)$*,*
- $com(M)$ *is a set of commands that define the behavior of the module, where:* $\forall w \in \ com(M) : w \triangleq [act]guard \rightarrow p_1 : u_1 + \ldots + p_m : u_m$.

A system contains n sub-parts where each one is described by a module and its relation is described by an algebraic expression. The supported algebraic expression in PRISM are:

1. M1||M2 : It is a parallel composition of modules. M1 and M2 synchronize only on actions occurring in both M1 and M2,

2. M1|||M2: asynchronous parallel composition of M1 and M2 (fully interleaved, no synchronization),
3. M1|$[a, b, ...]$|M2: restricted parallel composition of modules M1 and M2 (synchronizing only on actions from the set a, b,...),
4. M/a,b,... : hiding of actions a, b, ... in module M,
5. Ma¡-b,c¡-d,... : renaming of actions a to b, c to d, etc. in module M.

Finally, the system, containing n modules, is defined formally in the Definition 3.3.3.

Definition 3.3.3 (PRISM System)**.** *A PRISM model is a tuple* $P = (name_P, var(P), sys, M_1, \ldots, M_n)$ *where:*

- $var(P)$ *is a finite set of system variables,where* $var(P) = var(G)(\bigcup_{i=1}^{n} M_i)$
- sys *is an algebraic expression that defines the models' communication,*
- $M_1, \ldots, M_n$ *is a countable set of modules.*

3.4 The PRISM Language

PRISM language is composed by two basic elements: modules and variables. A model is composed by a set of interacting modules. Each module has a number of integer local variables of finite range of values. A model may also have global variables, which are common for all the modules. The state of a module is defined by the value of its local variables. A state of the model is defined by the value of the variables of all modules combined. A state transition in the model is defined by a variance of the values of the variables inside the model. The possible transitions are defined by a set of commands inside each module, which can happen in a synchronous or in a asynchronous way. A command is composed by a label, which defines whether the command is synchronous or not, a guard, which identifies a subset of the global state and one or more updates, each tied to a probability of occurrence and each corresponding to a possible transition of the model.

Algorithm 1 Example of PRISM code

```
mdp
**module** Counter
$count$ : $[0..5]$ **init** $1$;
[] $(count = 1) \rightarrow (count' = 3)$;
[] $(count = 3) \rightarrow 0.5 : (count' = 5) + 0.5 : (count' = 1)$;
[s1] $(count = 5) \rightarrow (count' = 0)$;
**endmodule**
**module** X
$v$ : $[0..1]$ **init** $0$;
[s1] $(v = 0) \rightarrow (v' = 1)$;
**endmodule**
```

The simple example in Algorithm 1 is useful to clarify the structure of the PRISM language. There we have a model, defined as a Markov Decision Process [5], [11] (MDP), composed by two modules: one called 'Counter', with one integer variable 'count' ranging in value from zero to five and starting at one, and the other called 'X', with one integer variable 'v' ranging in value from zero to one and starting at zero. The second part of a module definition is composed by the set of commands. Each command is written in the format []g$\rightarrow$ u, where 'g' is the guard and 'u' the updates; if the predicate 'g' holds true, then 'u' will execute. There are two commands marked with a label 's1'. This means these commands must be executed at the same time, when both their guards become true.

3.4.1 Additional Consideration

In our research we worked with PRISM version 3.3.1, as it was the latest release at the time we started this project. Since that version does not support Probabilistic Timed Automata [9] (PTA), which would be ideal for modeling TTE, we simulate time in our model with (MDP) combined to labeled statements synchronized to an incrementing counter that acts as the clock.

The use of MDP is interesting for modeling a TTE network because a MDP can describe both nondeterministic and probabilistic behaviors, making use of a mathematical framework for modeling decision situations where the outcome combines probabilistic, random and non-deterministic results.

Although it may seem like a contradiction to use a non-deterministic model to verify a protocol where all communications occur in a deterministic manner, it is important to note that TTE is built on top of Ethernet, which a non-deterministic protocol. We reckon one of the challenges of implementing TTE is to turn a non-deterministic protocol (Ethernet) into a deterministic one, therefore modeling the network in a non-deterministic model in a way that it becomes deterministic is an accurate way of representing TTE.

Chapter 4

Time-Triggered Ethernet Network Scenarios

In this chapter we intend to discuss our modeling of two TTE network scenarios (one a Standard TTE configuration and the other a Fault-tolerant TTE configuration)and present the logical representation of those scenarios.

Some important considerations about our work are:

- Both network scenarios are closed-world (number of nodes is limited and known a priori) and we always assume a fault-free, collision-free start up. Both are built using a Star-topology.
- PRISM language models are based on the states of a system. For this reason our TTE model is focused on the network's behavior following the TTE specification rather than the internal functionality of each component. Our intention is to model the relationship between the different components inside the network, rather than the components themselves.
- A message is a random number of packet transmissions where only the act of sending (and not the data) is relevant. Every packet must receive an acknowledgement,

with the exception of the last one where, in lack of an acknowledgement for a set period of time, the client will assume the packet was received successfully.

- Every hardware component that we are simulating in PRISM is given a probability for failure, respecting the number of failures admissible in each of the network scenarios (one failure for the first scenario and two failures for the second scenario). It is important to notice that even though the probabilities of hardware failure in our system are not based in real-world specifications, this makes no difference since our objective is to verify the protocol's reaction once the failures have occurred to determine which, if any, are its susceptibilities to faults.

- In our Fault-tolerant TTE scenario, the Guardian implements the Leaky-Bucket control function.

- In real life there are two possible scenarios for client failure: silent and non-silent. In silent failure the client will become mute from that point onwards and in non-silent failure the client will start sending garbage-data whenever it has the chance. In our model we only work with cases of non-silent failure of the clients.

- The channel between Switch and Server (Standard scenario) or between Switch and Switch (Fault-tolerant scenario) will never fail.

4.1 Network Scenarios

We build our scenarios around basic constructions of clients (Figure 4.1) and switches (Figure 4.2). These state-machines show the main functionality of each component in the network. These components are combined later to form scenarios A (Figure 4.3) and B (Figure 4.4). All the TTE specifics are inserted directly in the PRISM code in the form of labels and guards.

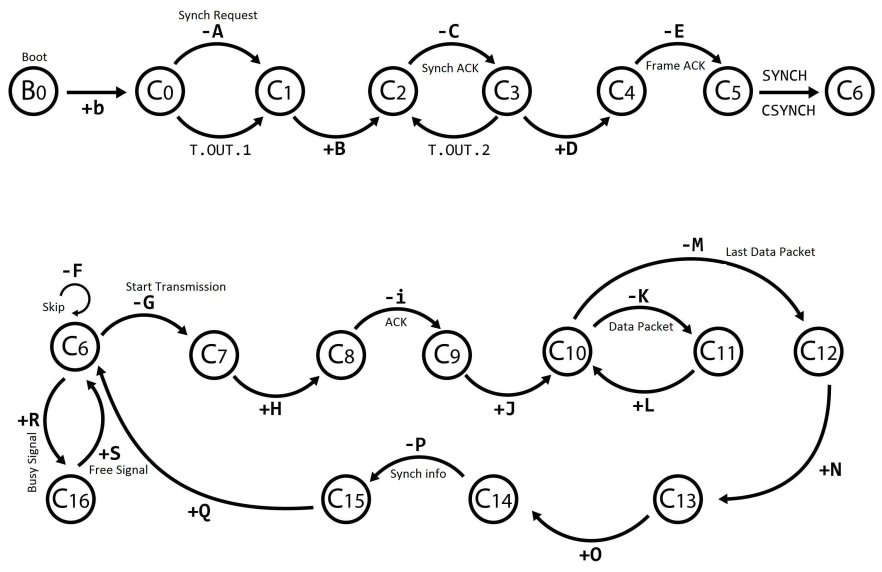


Figure 4.1: Host Computer (client) FSM

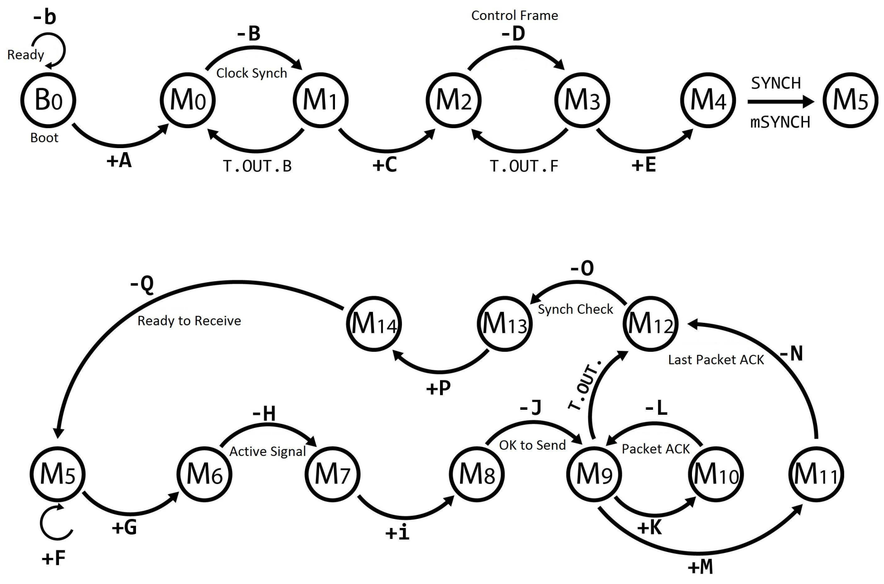


Figure 4.2: TTE Switch FSM

4.1.1 Standard TTE Scenario

Our first scenario describes a Standard TTE configuration, with three clients connected to a server through a single switch (Figure 4.3). Only the clients are able to send messages and the only possible destination is the server. With the exception of the server -including the wired connection between the switch and the server), hardware failures can be introduced in any element of the network (clients, switch or wiring) but since this scenario works with the assumption of Single-Fault Hypothesis, only one element may fail in each run. There's a back-up switch that will come online in case of catastrophic hardware failure of the main network switch and in case of wiring failure, only one connection will be affected. As mentioned previously, any case of client failure is a non-silent failure where the client will start flooding the network with garbage-data whenever it has the chance.

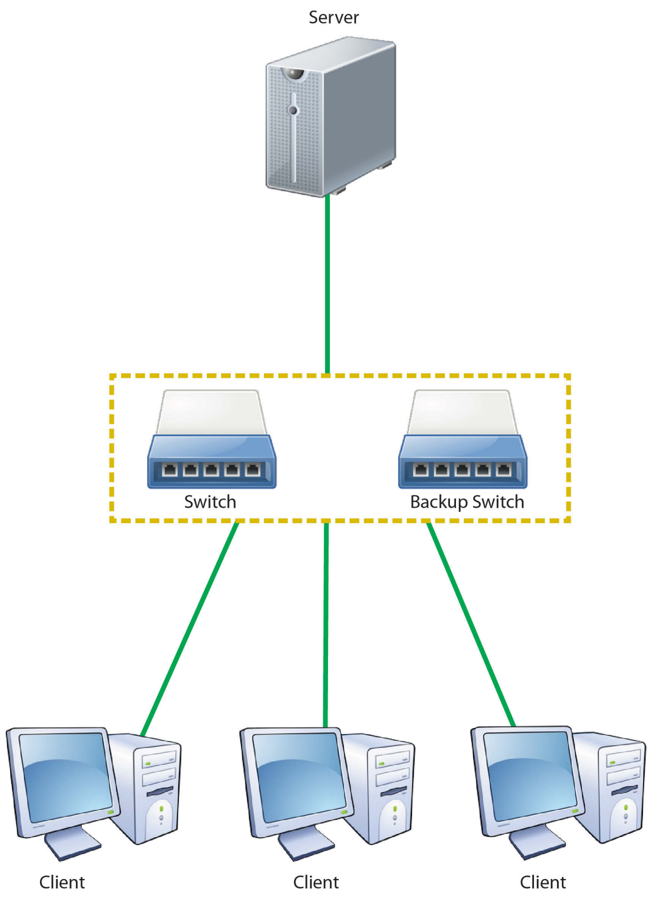


Figure 4.3: Topology of Standard TTE Network Scenario

This scenario uses a simple Master/Slave one-step synchronization protocol where the single master will calculate the synchronization clock, based on its local clock, and broadcast it to the rest of the network, thus we ignore the TTE Compression Function.

4.1.2 Fault-tolerant TTE Scenario

The second scenario runs under the same basic principles described in the first, but with some differences. This scenario simulates a safety-critical network (Figure 4.4) with double-failure hypothesis and a multi-master two-step clock synchronization, where the existence of more than one synchronization master requires an extra step for generating the synchronization clock. The multiple synchronization masters will send a control frame each to a compression master that will calculate an average value based on the latencies of those control frames as they arrive. Then the compression master sends a new control frame that will be used to synchronize the local clocks in the synchronization masters, as described in [3].

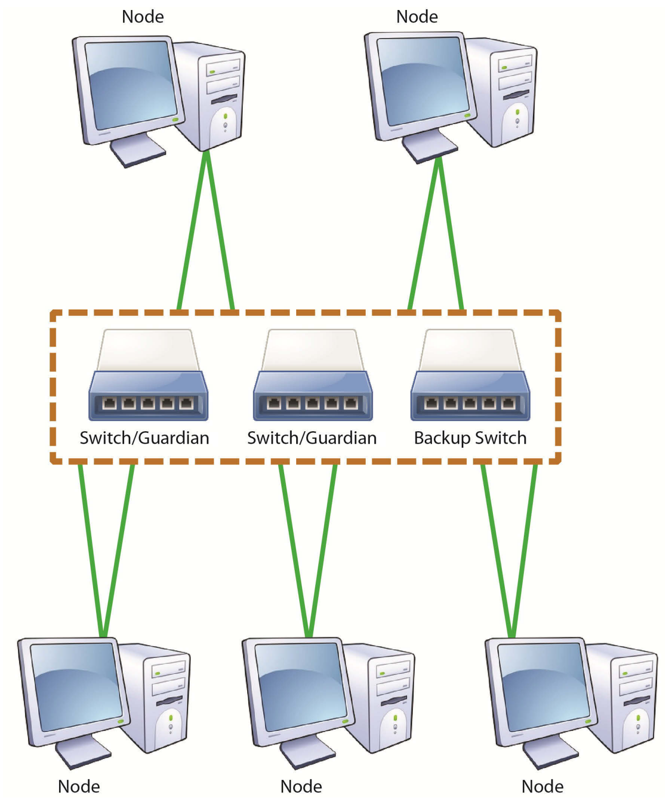


Figure 4.4: Topology of the Fault-tolerant TTE Network Scenario

The configuration consists of five host-computers (clients) and two switches/guardians on a star network configuration. The guardian uses a Leaky Bucket function. Every connection is redundant. There is a backup shadow-switch in case of catastrophic hardware failure of one of the original switches. In this scenario any two faults can happen at any given time with the only exception being that only one switch may fail per run since the scenario has only one backup and the safety-critical protocol requires at least two functioning switches.

4.2 Logical Model

In this section we explain the functionality of each component behind our TTE network models, namely the components that form the networks (client, switch, communication channel and clock). We use a form of PRISM pseudo-code to describe the modules and we explain them in text format.

Some parts of the original code have been replaced by logical analogous that make the code shorter and easier to understand, while maintaining its meaning. Some parts of the original code have been omitted because they intent to go around limitations in the expressibility of PRISM language; these parts will be explained in detail.

4.2.1 Clock Module

As mentioned previously, we used PRISM tool version 3.3.1 in this work. That version of PRISM does not natively support time. This was a very big problem in modeling a TTE network, as that protocol is built around the notion of Real-time transmission and time-division. Our solution for this issue was to build a PRISM module to act as a clock. This module consists on a counter that increments itself in a cyclical manner. This is the only module in our model that is not bound to any other modules, so it's commands are always free to execute. All other modules are bound by the Timer.

Algorithm 2 Pseudo-code for a PRISM Timer

```
1: module Timer
2: [] $(time = 0) \rightarrow (time' = 1)$;
3: [] $(time = 1) \rightarrow (time' = 2)$;
4: [] $(time = 2) \rightarrow (time' = 3)$;
5: [] $(time = 3) \rightarrow (time' = 0)$;
6: endmodule
```

Algorithm 2 exemplifies the functionality of Timer. The Timer module manipulates a variable 'time' which is a global integer variable in the model, with value range of *"n+1"* - *n* being the number of clients plus the number of switches in the network. Timer has no labels and its guards are built in a form that in any possible state of the model, always one of its guards will be true. It will run in an infinite loop from zero to *n+1* (this example assumes a network with two clients and one switch). This implementation ensures fault-tolerant global time, required by the TTE Synchronization Function. After the synchronization step all entities will obey this clock.

4.2.2 Transmission Channel

In our model we have one instance of Transmission Channel for every connection between a client and a switch. Its general function can be seen in Algorithm 3.

Algorithm 3 Pseudo-code for a TTE Communication Channel

```
1: module Channel 01
2: $ch1 : [0..3]$ init 0; -state of the channel.
3: $failure : [0..1]$ init 0; -indicates presence of failure in the channel.
4: if failure is 0 then
5:     [Msnd1] $(ch1 = 0) \rightarrow (ch1' = 1)$;
6:     [Crcv1] $(ch1 = 1) \rightarrow (ch1' = 2)$;
7:     [Csnd1] $(ch1 = 2) \rightarrow (ch1' = 3)$;
8:     [Mrcv1] $(ch1 = 3) \rightarrow (ch1' = 0)$;
9: end if
10: if failure is 1 then
11:     [Csnd1] $(ch1 = 0) \rightarrow (ch1' = 1)$;
12:     [Csnd1] $(ch1 = 1) \rightarrow (ch1' = 2)$;
13:     [Csnd1] $(ch1 = 2) \rightarrow (ch1' = 3)$;
14:     [Csnd1] $(ch1 = 3) \rightarrow (ch1' = 0)$;
15: end if
16: [] $(failure = 0) \rightarrow 0.05 : (failure' = 1) + 0.95 : (failure' = 0)$;
17: endmodule
```

The channel has two variables. The variable *ch* keeps track of the state of the channel. This is very important because of the deterministic nature of TTE. The second variable *failure* is used to implement a failure logic into the channel, as the name would suggest. As long as the channel is fault-free (*failure* = 0) it is susceptible to a chance of entering fail state of 0.05 percent.

When not in faulty state, the channel will respect the labels Msnd1, Crcv1, Csnd1, Mrcv1 (respectively 'master send', 'client receive', 'client send', 'master receive'). As mentioned previously, within a PRISM module, commands that bear labels can only be executed in synchrony with the commands of another module bearing the same label. This structure of the communication channel binds the communication in a deterministic form, that should become clearer once we reach the clients and the switches.

When the channel enters faulty state, all labels become ”client send” (Csnd1) allowing the client to freely send messages into the channel. It’s very important to note that for our model a failure in the communication channel means a failure in the client that’s connected to it. Having all labels as ”Csnd1” mean that the client is free to flood the network with messages (non-silent failure discussed previously).

4.2.3 Switch (Master)

In this module (as well as in the Client module) every transmission is guarded by the state of the Timer module and the transmission must be synchronized with the state of the Channel module. This structure ensures Deterministic Communication where no two messages can be sent at the same time, as the TTE protocol dictates.

This is also part of the Guardian Function of TTE that is responsible for fault tolerance. Our Guardian Function implements the ”leaky bucket” service, which limits the amount of data that a component is allowed to send. In PRISM we modeled the fault tolerance by restraining every send and receive action to a rendezvous connection (controlled by the synchronization labels). We eliminate this rendezvous requirement in case of hardware failure. By this method, even if one node is faulty its messages will not do any harm to the network and they will be ignored by the other nodes. in TTE terms it means that a fault or the errors generated by a fault will not be allowed to leave their containment zone.

Algorithm 4 describes the functionality of a Switch.

Algorithm 4 Pseudo-code for a TTE Switch

```
1:  module Switch 01
2:  $Mboot$ : $bool\ init\ true$; -variable that shows if the switch has yet to boot.
3:  $rcv$ : $[0..20]$ $init$ 0;
4:  $snd$ : $[0..20]$ $init$ 0; -snd and rcv store the state of the transmission.
5:  [Msnd1] $(time = 0)\ AND\ (Mboot = true) \rightarrow (snd' = 1)\ AND\ (Mboot' = false)$; -snd
    state 1 means the master/switch is making the first broadcast in the network.
6:  [Mrcv1] $(msg = 1) \rightarrow (rcv' = 1)$; -received acknowledgement(Ack).
7:  [Msnd1] $(time = 0)\ AND\ (rcv = 1)) \rightarrow (snd' = 2)$; -Sending Clock Synch.
8:  [Mrcv1] $(snd = 2) \rightarrow (rcv' = 2)$; -Master receives clock Ack.
9:  [Msnd1] $(time = 0)\ AND\ (rcv = 2) \rightarrow (snd' = 3)$; -Master sends control frame.
10: [Mrcv1] $(snd = 3) \rightarrow (rcv' = 3)$; -Receives Ack.
11: [] $(rcv = 3)\ AND\ (help = 0) \rightarrow (synch' = true)\ AND\ (snd' = 4)$; -Synchronization
    Happened.
12: [] $(synch = true) \rightarrow (busy' = 0)$; -Master ready to receive data.
13: [Mrcv1] $(busy = 0)\ AND\ (Snode = 0)\ AND\ (rcv = 3)\ AND\ (help = 1) \rightarrow (rcv' = 4)$;
14: [] $(rcv = 4) \rightarrow (busy' = 1)$; -On a transmission.
15: [Msnd1] $(time = 0)\ AND\ (rcv = 4)\ AND\ (busy = 1) \rightarrow (snd' = 5)\ AND\ (rcv' = 5)$;
    -Master informs the node that the connection has been established.
16: [Mrcv1] $(rcv = 5) \rightarrow (rcv' = 6)$; -received Ack
17: [Msnd1] $(time = 0)\ AND\ (rcv = 6) \rightarrow (snd' = 6)$; -Send to the client Permission to
    send.
18: data loop start
19: [Mrcv1] $(snd = 6)\ AND\ (EoT = false) \rightarrow (rcv' = 7)$; -Received data packet.
20: [Mrcv1] $(snd = 6)\ AND\ (EoT = true) \rightarrow (rcv' = 8)$; -Received last packet.
21: [Msnd1] $(time = 0)\ AND\ (rcv = 7) \rightarrow (snd' = 7)$; -Sending packet Ack.
22: [] $(snd = 7) \rightarrow (snd' = 6)$; -Adjustment to stay in loop.
23: [Msnd1] $(time = 0)\ AND\ (rcv = 8) \rightarrow (snd' = 8)\ AND\ (rcv' = 9)$; -Ack last packet.
24: data loop end
25: [Msnd1] $(time = 0)\ AND\ (snd = 8) \rightarrow (snd' = 9)$; -Free signal.
26: [Mrcv1] $(snd = 9) \rightarrow (rcv' = 10)$; -Receives Alive message from idle clients.
27: [] $(rcv = 10) \rightarrow (EndProgram' = true)$; -In this example the switch stops after a
    successful transmission.
28: endmodule
```

The way the switch works is this: after booting it will synchronize with the client through a dedicated channel. After that the client may or may not send data according to its own logic. We can define the number of clients allowed to transmit at the same time but we kept that number limited to one because our tests showed that aside from a significant increase in the total number of states, increasing the number of clients allowed to transmit had no impact on the verification.

This process sends and receives from the channel and it is guarded by both time and channel state. We modeled all different states in this process (and also in the Client module) bear different names. This was done to ensure the TTE property *naming*. That property says that TTE has a naming scheme that's able to identify all message types supported by it.

Since real data transfer between variables and modules is not supported by PRISM we ignore the TTE property **message format**; as mentioned previously we only care for the action of sending but we do not care about what's being sent. Still we take this property into consideration when we model the 'sending Control Frame' step.

This feature is now shown in the algorithm above but a Switch has also a fault logic. Similarly to how he handle faults in the Channel, the Switch has a boolean variable that marks the occurrence of hardware failure. When failure happens, an additional guard will block all commands in the module permanently and a Shadow Switch will boot to take the place of the failing Switch.

4.2.4 Client

The Client module, like the Switch before it, is constructed around the principles and boundaries of the Channel module detailed previously.

Every client has its own ID number and when it is transmitting it will inform the network that its ID is active (through the *Snode* variable which is global). This allows every message to be traced back to its source ensuring that the system meets the *Consistent Diagnosis* TTE property.

Every client has a logic for entering the "non-silent fault-state" after a failure is triggered on the Channel. The logic resembles what happens in the Channel module, where the code is repeated without any control labels, which will cause it to ignore the rendezvous guards and overflow the network with data. The whole fault logic will be omitted to avoid repetition of code.

Algorithm 5 Pseudo-code for a TTE Client

```
1: module Client 01
2: $Cboot$ : $bool\ init\ true$; -variable that shows if the client has yet to boot.
3: $wait$ : $bool\ init\ false$; -Signals if client is in idle state.
4: $rcv$ : $[0..20]$ $init$ 0;
5: $snd$ : $[0..20]$ $init$ 0; -snd and rcv store the state of the transmission.
6: $ID$ : $[1..2]$ $init$ 1; -This is the number of the client. Each will have a different ID
7: [Crcv1] $(Cboot = true) \rightarrow (rcv' = 1)\ AND\ (Cboot' = false)$; -rcv 1 means Client
   received master's broadcast after booting.
8: [Csnd1] $(time = 2)\ AND\ (rcv = 1) \rightarrow (snd' = 1)$; -snd 1 = synch token.
9: [Crcv1] $(snd = 1) \rightarrow (rcv' = 2)$; -Receive Clock synch.
10: [Csnd1] $(time = 2)\ AND\ (rcv' = 2) \rightarrow (snd' = 2)$; -msg 2 is a synch ack.
11: [Crcv1] $(snd' = 2) \rightarrow (rcv' = 3)$; -Receive control frame.
12: [Csnd1] $(time = 2)\ AND\ (rcv = 3) \rightarrow (snd' = 3)\ AND\ (rcv' = 4)$; -Send ack.
13: [] $(snd = 3)\ AND\ (synch = true) \rightarrow 0.3 : (snd' = 4)\ +\ 0.7 : (snd' = 5)$; -Client may
    request to send a message(5) or skip its turn(4).
14: [] $(snd' = 4)\ AND\ (synch = true) \rightarrow 0.3 : (snd' = 4)\ +\ 0.7 : (snd' = 5)$; -Same as
    above for a client who has skipped its turn before..
15: [Csnd1] $(time = 2)\ AND\ (snd = 5)\ AND\ (busy = 0) \rightarrow (snd' = 6)$; -If client has a
    message it will send a request (6).
16: [] $(snd = 6)\ AND\ (synch = true) \rightarrow (synch' = false)\ AND\ (help' = 1)$; -
    Conformation rule. needed to set synch to false before next Mrcv.
17: [] $(snd = 5)\ AND\ (busy = 1) \rightarrow (wait' = true)$; -When this command is activated only
    one client each turn.
18: [Crcv1] $(snd = 6)\ AND\ (help = 1) \rightarrow (rcv' = 7)\ AND\ (snd' = 7)$;
19: [] $(rcv = 7) \rightarrow (Snode' = ID)\ AND\ (rcv' = 8)$; -Receive active node status.
20: [Csnd1] $(time = 2)\ AND\ (rcv = 8) \rightarrow (snd' = 8)$; -Send ack.
21: [Crcv1] $(snd = 8) \rightarrow (rcv' = 9)$; -Receive Permission to Send.
22: data loop start
23: [Csnd1] $(time = 2)\ AND\ (rcv = 9) \rightarrow 0.9 : (snd' = 9)\ +\ 0.1 : (snd' = 10)$; -Sending:
    msg 9 = packet. msg 10 = last packet.
24: [] $(snd = 10) \rightarrow (EoT' = true)\ AND\ (snd' = 11)$; -last packet sent = end of transmission
    (EoT).
25: [Crcv1] $(snd = 9) \rightarrow (snd' = 0)$; -Data packet Ack.
26: [Crcv1] $(snd = 11) \rightarrow (rcv' = 11)$; -Last Data packet Ack.
27: data loop end
28: [] $(rcv = 11) \rightarrow (wait' = true)\ AND\ (Snode' = 5)$; -Client goes idle after the end of
    transmission.
29: endmodule
```

Our model do not work with *Legacy Integration*. Although it is not implemented the current structure would be able to handle legacy connections with just a few minor adjustments.

Our model can handle *Predictable and Deterministic message transfer, strong fault isolation, scalability, seamless communication and no single point of failure*. All the switches are designed following this synchronous channel model in order to ensure those properties listed above. With that, all the properties required for a TTE network are dealt with.

Chapter 5

Verification and Results

The way we verify the properties of our model in PRISM is closely related to our methodology for generating counter-examples. In this chapter we derive and explain our properties in PCTL language and we explain how we generate counter examples based on the output of PRISM simulation.

5.1 Property Specification

In our scenarios we are verifying a set of eight properties that, if held true, will guarantee the network is running according to TTE specifications.

Following are the PCTL formulas, having always Client "A" in mind like previously. The variables in the formulas represent actual states in our model; please refer to figures 4.1 and 4.2 for details.

1. Property - The transmission only starts after successful synchronization of the node.
   Formula - Pmax=? [G ($\neg(C6_A \wedge M5_A) \implies \neg(F(C10_A \wedge M9_A \wedge (Time_A = 1)))$)];

2. Property - A transmission only occurs if all nodes are synchronized.
   Formula - Pmax=? [G ($\neg(C6_A \wedge M5_A) \vee \neg(C6_B AND M5_B) \vee \ldots \vee \neg(C6_n \wedge M5_n) \implies$

$\neg(F(C10_A \wedge M9_A \wedge (Time_A = 1))))$];

3. Property - At any given point in time there's only one node sending a message.
   Formula - Pmax=? [G $((C10_A \wedge M9_A \wedge (Time_A = 1)) \implies \neg(C10_B \wedge M9_B \wedge (Time_B = 2)) \wedge \ldots \wedge \neg(C10_n \wedge M9_n \wedge (Time_n = N)))$];

4. Property - A node only sends on its time partition.
   Formula - Pmax=? [G $(C10_A \wedge M9_A \wedge (Time_A = 1) \wedge \neg(C10_A \wedge M9_A \wedge (Time_A = 2)) \wedge \ldots \wedge \neg(C10_A \wedge M9_A \wedge (Time_A = N))$)];

5. Property - Every node has its own unique time partition.
   Formula - Pmax=? [G $((Time_A = 1) \wedge (Time_A \neg = Time_B) \wedge \ldots \wedge (Time_A \neq Time_n))$];

6. Property - A sent packet will be received by the destination.
   Formula - Pmax=? [G $(C10_A \implies F(C11_B \vee C12_B))$];

7. Property - To every message sent there would be an acknowledgement.
   Formula - Pmax=? [G $((C11_A \implies FC10_A) \vee (C12_A \implies FC13_A)))$];

8. Property - Every transmission will eventually end.
   Formula - Pmax=? [G $(C10_A \implies FC13_A)$];

After running the verification with the properties mentioned above, we re-wrote the same eight properties with the use of model flags to mark certain relevant states. This was done to hold a comparison between the two sets of properties as for their syntax and their performance during verification. The properties in both lists correspond by number.

The way we perform this second verification of the protocol involves creating a set of state-triggered flags (Boolean variables) in the PRISM code. Whenever a certain state is reached, a specific flag will become true to mark that event (like entering transmission mode) and, whenever applicable, if that event is no longer true the flag will be set to false (like exiting transmission mode). We, then, use a set of assertions about these flags, that should always hold true during the verification.

The use of this method of identifying specific states by the use of flags greatly simplifies the process of writing the PCTL properties and it is a major component of our methodology for generating counter-examples. This method is superior to simply generating PRISM labels from model states because labels can only be used for formal verification; they cannot be traced or manipulated in simulation mode.

Following we give the PCTL formulas, having always Client "A" in mind (note that for the verification we have a similar set of assertions for each Client):

1. Property - The transmission only starts after successful synchronization of the node.
   Formula - Pmax=? [G $\neg(\neg Synch_A \wedge Tr_A)$];
   Where "Synch" represents that the node has successfully synchronized with the network and "Tr" shows that a transmission is in process. The letter after each variable identifies the node.

2. Property - A transmission only occurs if all nodes are synchronized.
   Formula - Pmax=? [G $\neg(Tr_A \wedge (\neg Synch_A \vee \neg Synch_B \vee \ldots \vee \neg Synch_N)$];

3. Property - At any given point in time there's only one node sending a message.
   Formula - Pmax=? [G $(Tr_A \implies (\neg Tr_B \wedge \neg Tr_C \wedge \ldots \wedge \neg Tr_n))$];

4. Property - A node only sends on its time partition.
   Formula - Pmax=? [G $(Tr_A \implies Time_A)$];
   Where "$Time_A$" is a flag that becomes true when the system clock enters the time partition assigned to node A.

5. Property - Every node has its own unique time partition.

   Formula - Pmax=? [G $(Tp_A \wedge Tp_B \wedge \ldots \wedge Tp_n)$];

   Where ”Tp_A” is a flag that becomes true only if ”$Time_A$” is different than the ”Time” of all other nodes.

6. Property - A sent packet will be received by the destination.

   Formula - Pmax=? [G $(Snd_A \implies F\ Rcv_B)$];

   Where ”Snd” represents that a packet has been sent and ”Rcv” represents that a packet has been received.

7. Property - To every message sent there would be an acknowledgement.

   Formula - Pmax=? [G $(Snd_A \implies F\ Ack_A)$];

   Where ”Ack” represents that an acknowledgement message has arrived.

8. Property - Every transmission will eventually end.

   Formula - Pmax=? [G $(Tr_A \implies F\ Tend_A)$];

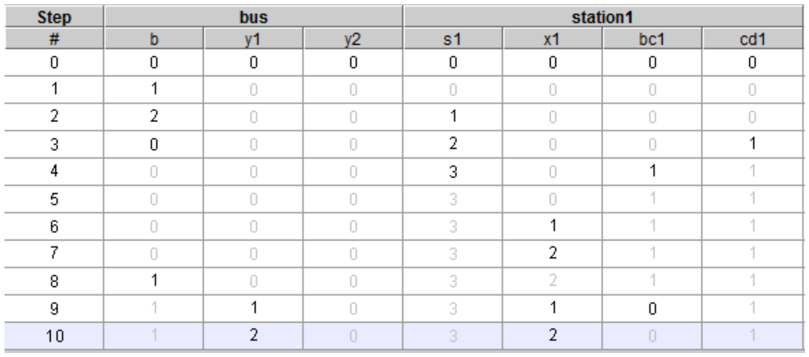
   Where ”Tend” is a flag that becomes true when the transmission ends.

The decrease in complexity in all properties (excluding number 8, for design reasons) is notable. We talk about the results of the verification and express a comparison betweens the two sets of properties in section 5.3.

5.2 Counter-Example Generation

To generate counter-examples from a property fail we take advantage of the flag logic mentioned before, where boolean flags are used to identify when the model reaches a determined state. We use the same logic behind the creation of Labels in PRISM, except that instead of assigning a relation of verification-relevant states to a Label, we use those states as guards for a boolean variable that will act as a flag.

PRISM model checker has a built-in simulator where the user can automatically generate a number of random steps as well as which variables from the system he wants to monitor and PRISM will run a simulation, returning the value of each of the selected values in every step in form of a table. Figure 5.1 shows a small example of what the simulation table looks like.



Step	bus			station1			
#	b	y1	y2	s1	x1	bc1	cd1
0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
2	2	0	0	1	0	0	0
3	0	0	0	2	0	0	1
4	0	0	0	3	0	1	1
5	0	0	0	3	0	1	1
6	0	0	0	3	1	1	1
7	0	0	0	3	2	1	1
8	1	0	0	3	2	1	1
9	1	1	0	3	1	0	1
10	1	2	0	3	2	0	1

Figure 5.1: Standard TTE Configuration

Our method consists in first running PRISM formal verification on the model. In case of a property fail, we switch PRISM into simulation mode and we generate a very extensive simulation of the model (for our model we used a simulation with 500.000 steps generated randomly). Then we extract the simulation table from PRISM in a text file (*.txt extension) and we perform a heuristic search on the table inside the text file.

For this purpose we have created a tool in C++ language. Our tool takes as input the property that failed PRISM verification and the text file with the simulation table extracted from PRISM. The software performs an Artificial Intelligence search technique known as Hill Climbing [19] which consists in a heuristic iterative search method designed to improve efficiency in searching very large state-spaces. It scans the simulation table, looking for the negation of the property specified as input. The state-space of the search is based on the state flags defined into the PRISM model.

If the tool is successful in finding a state where the negation of the input property is true, then it will exit the search and prompt the user for input. At this point the user is able to see the values of all variables that are part of the failing state and he can extract that information into a new text file in form of a table of values. He can also specify how many steps from the source table he wishes to extract. For example if the user chooses to extract ten steps, the program will generate a table containing the failing state plus the ten previous steps leading to the failing state. This allows the user to trace what path in the model leads to the failing state for that given property.

By this method we were able to trace and identify the circumstance that will generate a state that violates the TTE properties.

5.3 Results

In this section we seek to discuss the results of our work in more detail, starting with a comparison between the verification of both sets of properties introduced in section 5.1. After this we discuss the results of the verification, including the description of the circumstances that lead to the occurrence of the faulty state and we finish this chapter by making a brief stochastic analysis of the network scenario B, where we run several experiments with different failure probabilities for the network elements.

5.3.1 Verification and Counterexample

In this section we explain how we performed the verification of our models and how we generate counterexamples for failing states. Table 5.1 brings some relevant data about the models of our scenarios.

Table 5.1: Scenario Models.

Feature	States	Nodes	Transitions	Decisions	Time(s)
Scenario A	3459	10299	8108	7502	0,2133
Scenario B	764533	4×10^8	6.72×10^9	5.58×10^9	12,49

As it can be seen on the table above, the total number of states, nodes, transitions and decisions in scenario B is huge. In fact, scenario B suffers from state-explosion and PRISM cannot even compute the reachable model for it in its complete form. We were able to generate the numbers in Table 5.1 by simplifying scenario B (by editing the code of scenario B to lock three of the five clients in 'wait' state and establishing a communication between the other two). Even by doing this, the size of the reachable model is astronomical. Verifying scenario B was only possible by partitioning the code into sub-models. The procedure we took was to remove the synchronization phase and the data transfer phase from the model. With this, the network is now divided into three different PRISM models, each one in a different file. After this we build and verify each one of them separately. For the purpose of this division, the synchronization phase starts from boot and it ends when the client tries to start a transmission. The data transfer phase is the data loop and the communication that takes place between synchronization and data transfer and after data transfer form the third sub-module. These markers where the code was divided are specified in the algorithms in Section 5.1.

In the first verification run for both scenarios we suppressed the possibility of hardware failure, assuming every component would always function under ideal conditions. We were able to successfully verify both scenarios under these settings and all the properties passed. Then we run a second verification for each scenario allowing hardware failure to happen in a probabilistic manner and using PRISM and our counter-example tool we were able to find one state in the second scenario where some properties do not hold true.

The faulty state in the second scenario violates the properties that say that "at any given point in time there's only one node sending a message" (property number 3) and "a node only sends on its time partition" (property number 4). The faulty state may be reached when one of the switches and one of the nodes fail at the exact same time. Whenever a switch fails, communication will be frozen and the backup switch will boot and the synchronization function will synchronize the clocks of both switches and then broadcast the new clock-synch to the entire network and allowing communication to resume. In this particular case where the client fails at the same time as the switch, there is a chance that the random data from the client's non-silent failure will collide with the message from the switch, making the failure to pass unnoticed by the guardians. After this, there is a chance that the second garbage-message from the failing client will collide with the new clock-synch coming from the switches and on top of that, there's a probability that another node will be assigned to the same time partition that is being used by the failing client. When this state is reached, we'll have two nodes assigned to the same time partition, plus one failing node which is not yet known to the network's guardians.

Tables 5.3 and 5.2 bring the verification times and results for all eight properties. Both tables refer to network scenario B with probability of hardware failure. The tables show the verification time for each property and if the property was verified ("V") or if it failed ("F").

Following are the first eight properties specified in Section 5.1.

Table 5.2: Extended Properties.

Property	1a	2a	3a	4a	5a	6a	7a	8a
Time(s)	0.47	0.49	0.43	0.35	0.36	0.29	0.41	0.29
Result	V	V	F	F	V	V	V	V

Table 5.2 shows the results of the verification for the second set of properties specified in Section 5.1.

Table 5.3: Simplified Properties

Property	1b	2b	3b	4b	5b	6b	7b	8b
Time(s)	0.33	0.33	0.31	0.29	0.30	0.25	0.35	0.28
Result	V	V	F	F	V	V	V	V

As it can be noted, the properties written following the flagged-state methodology were verified faster every time. The verification results were the same, as it would be expected.

5.3.2 Stochastic Analysis of the Network

It is important to note that while PRISM could be used to accurately calculate the probability of a failing state being reached, for our purposes it was more important to identify the faulty state rather than finding its accurate probability of happening in the real world. For this reason we used probabilities of failure that do not reflect reality in our model.

We plotted the results of failure rates pertaining Clients and Switches in network Scenario B, bearing different failure probabilities, in order to compare and analyze them. In figure 5.2 it is possible to see how likely the different clients of scenario B are to enter failing state. All nodes have different failure probabilities and their likeness to fail is given in function of transmission round (K):

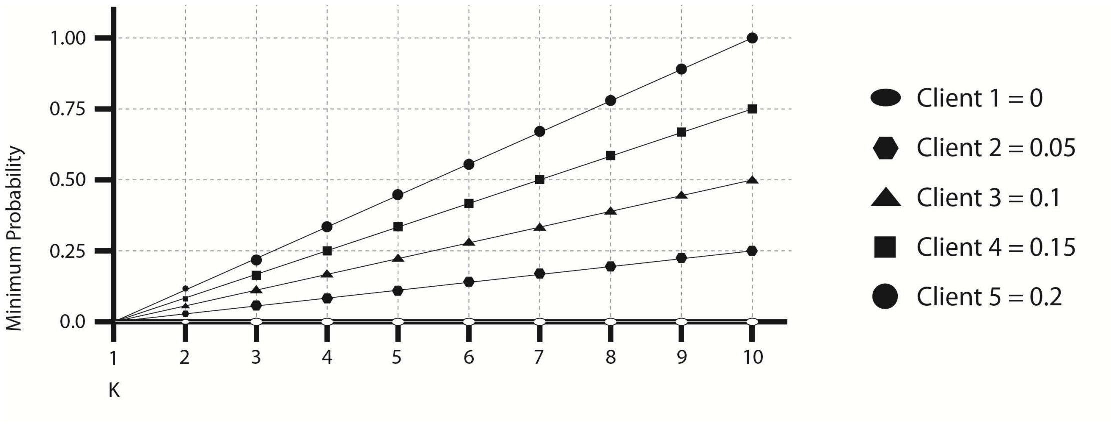


Figure 5.2: Client Failure Rates

In this test "Client 1" is the control node with no no chances of failure. The client's failure probability increase at a constant rate of 0.05%. The progression of the failure rate is linear for all clients due to the deterministic nature of TTE.

In Figure 5.3 we do the same thing as above but taking the failure probabilities of the network switches.

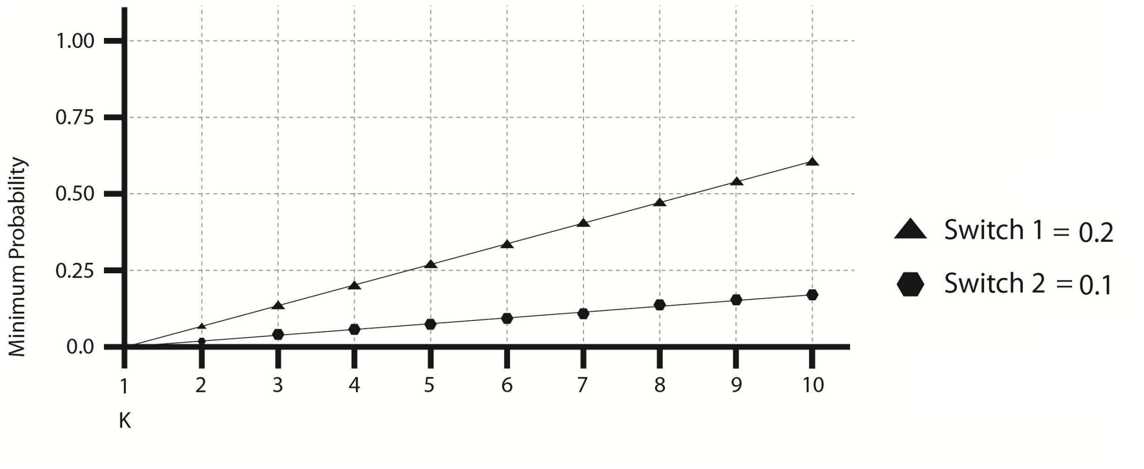


Figure 5.3: Switch Failure Rates

The switches show a similar behavior to the clients as it should be expected, however they are less likely to fail during a transmission cycle due to their reduced number as well as their failure limit (in our model only one switch may fail in contrast to two possible failures on the clients side).

As mentioned before, we always use closed-world scenarios where all elements are known and defined a priori. The reason for this is the limitation of the PRISM language, which does not support the design of open and dynamic models. We attempt to add some level of scalability to our model by writing our PRISM code divided in several processes (modules), detailed in Chapter 2, and making them as close as possible to the Object Oriented programming paradigm [23], instead or writing the whole code as one single block. In other words our model consists of several PRISM processes, each uniquely identifiable and performing a specific role in the network. Thanks to this approach we can easily add more elements to the network with some small adjustments to the code. As an example, to add a new node to the network all there is to do is to create a new ”Client” process with a new unique ID, add the channels that represent its network connections and update the new number of existing clients in the network switches.

Chapter 6

Conclusion and Future Work

We have proposed a model and verified two deterministic real-time network scenarios implementing Time-Triggered Ethernet protocol, using PRISM Model checker to build a high-level software implementation. Our approach is based on MDP model, which is the most accurate to represent a TTE network on the PRISM tool and we were able to identify one circumstance where TTE's properties will not be respected, likely resulting in catastrophic network failure. To our knowledge this is the first time this kind of verification is done on TTE protocol.

On top of that we have created a tool to generate counter-examples based on the results of PRISM simulation that, in theory, can be used to determine the path that leads to a failure in any model coded in compliance with our flagged-state methodology. Some research on other works based on PRISM Model Checker showed that the idea of using flags to identify relevant states and then running the verification based on these flags is not a common technique among PRISM designers. The absolute majority of people using PRISM adopt labels for their verification and while this is an easier method, labels only exist during verification and their values cannot be exported nor experimented with.

Our work can be used as a reference to further improve the reliability on the already robust TTE protocol and it can be extended by verifying different settings and implementations of that protocol or by going deeper into its many modules and verifying their functionality in detail. We would also like to extend our model to the newest version of the PRISM tool [14] which has true support for timed models such as Probabilistic Timed Automata (PTA) and Priced Probabilistic Timed Automata (PPTA), for better representation of a real-world scenario.

During the course of this research we have learned that TTE protocol is being implemented in some hospitals in North and South America and in Europe in the field of Computer-Assisted Surgery. TTE is also used by NASA for reliable aerospace communication. This greatly adds to the importance of this research and its findings and creates a great deal of motivation to continue on this path and expand what we have done.

Regarding our tool, we would like to automate it and improve the interactivity, adding an end-user interface to it and allowing the user to visualize and navigate through the states in the network to better understand the point of failure and why does it happen. Furthermore it is possible to modify the code of the PRISM tool in order to establish a direct communication between PRISM's simulation and our tool. With this the tool can receive a direct feed of the states generated by PRISM and upon identification of a state fit of the specifications it is looking for, the tool can issue a command to make PRISM stop the simulation, making the process fully automated.

6.1 Appendix

Code Sample for Network Scenario B in its full form.

```
mdp   //



global synch1 : bool init false;
global synch2 : bool init false;
global synch3 : bool init false;
global synch4 : bool init false;
global synch5 : bool init false;
global busy : [0..2] init 2;
global busy2 : [0..2] init 2;
global Snode : [0..6] init 0; // 0 = free.  6 = free after transmission
global help : [0..2] init 0;
global EoT : bool init false; // end of transmission
global EndProgram : bool init false;  // End of the execution.
global failS1:[0..1] init 0;
const int iID = 1;
const int iID2 = 2;
const int iID3 = 3;
const int iID4 = 4;
const int iID5 = 5;
const int iTimeA = 2;
const int iTimeB = 3;
const int iTimeC = 4;
const int iTimeD = 5;
const int iTimeE = 6;

// timer variables
global time : [0..7] init 0;

module Timer

[](time = 0) -> (time'=1);
[](time = 1) -> (time'=2);
[](time = 2) -> (time'=3);
[](time = 3) -> (time'=4);
[](time = 4) -> (time'=5);
[](time = 5) -> (time'=6);
[](time = 6) -> (time'=7);
[](time = 7) -> (time'=0);

endmodule



module Channel_A

ch1:[0..3] init 0;
fail1: [0..1] init 0;
[Msnd] (ch1=0)&(fail1=0) -> (ch1'=1);
[rcvC] (ch1=1)&(fail1=0) -> (ch1'=2);
[Csnd] (ch1=2)&(fail1=0) -> (ch1'=3);
[rcv] (ch1=3)&(fail1=0) -> (ch1'=0);
```

```
[Csnd] (ch1=0)&(fail1=1) -> (ch1'=1);
[Csnd] (ch1=1)&(fail1=1) -> (ch1'=2);
[Csnd] (ch1=2)&(fail1=1) -> (ch1'=3);
[Csnd] (ch1=3)&(fail1=1) -> (ch1'=0);

[] (fail1 = 0) -> 0.05:(fail1'=1) + 0.95:(fail1'=0);
[] (synch1 = true)&(ch1 = 0) -> (ch1'=2);
//conformation rule because the client must send twice in a row, before and after synch point.
[] (EoT = true)&(ch1 = 1) -> (ch1'=0)&(EoT'=false);
// adjustment so the master will send twice in a row after the end of transmission
endmodule

module
Channel_B=Channel_A[ch1=ch2,fail1=fail2,Msnd=Msnd2,rcv=rcv2,Csnd=Csnd2,rcvC=rcvC2,synch1=synch2]
endmodule
module
Channel_C=Channel_A[ch1=ch3,fail1=fail3,Msnd=Msnd3,rcv=rcv3,Csnd=Csnd3,rcvC=rcvC3,synch1=synch3]
endmodule
module
Channel_D=Channel_A[ch1=ch4,fail1=fail4,Msnd=Msnd4,rcv=rcv4,Csnd=Csnd4,rcvC=rcvC4,synch1=synch4]
endmodule
module
Channel_E=Channel_A[ch1=ch5,fail1=fail5,Msnd=Msnd5,rcv=rcv5,Csnd=Csnd5,rcvC=rcvC5,synch1=synch5]
endmodule

module Switch1

Mrcv:[0..20];
Mmsg:[0..20];
Mboot : bool init true;

[Msnd]((time = 0)|(time =1))&(Mboot = true) -> (Mmsg'=1)&(Mboot'=false);
// Mmsg 1 = "I'm alive" broadcast
[rcv](Mmsg = 1) -> (Mrcv'=1);
// If I sent alive broadcast I will receive answers from clients
[Msnd]((time = 0)|(time =1))&(Mrcv = 1) -> (Mmsg'=2);
// Sending Clock Synch
[rcv](Mmsg = 2) -> (Mrcv'=2);
// Master receives clock ack
[Msnd]((time = 0)|(time =1))&(Mrcv = 2) -> (Mmsg'=3);
// Master sends control frame
[rcv](Mmsg = 3) -> (Mrcv'=3);
// Receives Ack

[](Mrcv = 3)&(help = 0) -> (synch1'=true)&(Mmsg'=4);
//synch
[](synch1 = true) -> (busy'=0);
// Master ready to receive data.

[rcv](busy = 0)&(Snode=0)&(Mrcv = 3)&(help = 1) -> (Mrcv'=4);
[](Mrcv = 4) -> (busy'=1);
[Msnd]((time = 0)|(time =1))&(Mrcv = 4)&(busy = 1) -> (Mmsg'=5)&(Mrcv'=5);
// Send Active node + Busy signal.
[rcv](Mrcv = 5) -> (Mrcv'=6);
// received ack
[Msnd]((time = 0)|(time =1))&(Mrcv = 6) -> (Mmsg'=6);
// Send to the client Permission to send.
// data loop start
[rcv](Mmsg = 6)&(EoT = false) -> (Mrcv'=7);
// Receive packet.
[rcv](Mmsg = 6)&(EoT = true) -> (Mrcv'=8);
// Receive last packet.
[Msnd]((time = 0)|(time =1))&(Mrcv = 7) -> (Mmsg'=7);
// Ack packet.
[](Mmsg = 7) -> (Mmsg'=6);
// adjustment to stay in loop
[Msnd]((time = 0)|(time =1))&(Mrcv = 8) -> (Mmsg'=8)&(Mrcv'=9);
// Ack last packet.
// data loop end

[Msnd]((time = 0)|(time =1))&(Mmsg = 8) -> (Mmsg'=9);
// Free signal.
[rcv](Mmsg = 9) -> (Mrcv'=10);
// Receives Alive message from client.

[](Mrcv = 10) -> (EndProgram'=true);
// Here everything should go back as it was on synch state. Instead I'm ending the program.

[] (failS1 = 0) -> 0.05:(failS1'=1) + 0.95:(failS1'=0);
[] (failS1 = 1) -> (Mmsg'=0)&(Mrcv'=0);

endmodule

module Switch2 = Switch1 [ Mrcv=Mrcv2, Mmsg=Mmsg2, Mboot=Mboot2, rvc=rcv2, busy=busy2  ] endmodule

module Client1

Cmsg:[0..20];
// messages sent by client.
Crcv:[0..20] init 0;
// messages received by client.
Cboot : bool init true;
wait : bool init false;
// idle state.
ID:[0..5] init iID;
[rcvC](Cboot = true) -> (Crcv'=1)&(Cboot' = false);
// rcvd 1 = Client received master's broadcast. If I boot I will receive master's broadcast
[Csnd](time = iTimeA)&(Crcv = 1) -> (Cmsg'=1);
// Cmsg 1 = synch token. If the client does not answer it may generate deadlock so I excluded the option
[rcvC](Cmsg = 1) -> (Crcv'=2);
// Receive Clocl synch. This does not do anything because in this model the synch is fixed.
[Csnd](time = 2)&(Crcv = 2) -> (Cmsg'=2);
// Cmsg 2 is a synch ack.
[rcvC](Cmsg = 2) -> (Crcv'=3);
// receive control frame.
[Csnd](time = iTimeA)&(Crcv = 3) -> (Cmsg'=3)&(Crcv'=4);
// Send ack
//synch

[](Cmsg = 3)&(synch1 = true) -> 0.3:(Cmsg'=4) + 0.7:(Cmsg'=5);
// Client may request to send a message or skip its turn.
[](Cmsg = 4)&(synch1 = true) -> 0.3:(Cmsg'=4) + 0.7:(Cmsg'=5);
// Same as above. This eliminates deadlock.

[Csnd](time = iTimeA)&(Cmsg = 5)&(busy = 0) -> (Cmsg'=6);
```

```
// If client has a message and master is free, client will send a request.

[](Cmsg = 6)&(synch1 = true) -> (synch1'=false)&(help'=1);
//conformation rule. needed to set synch to false before next Mrcv

[](Cmsg = 5)&(busy = 1) -> (wait'=true);

[rcvC](Cmsg = 6)&(help = 1) -> (Crcv'=7)&(Cmsg'=7);
[](Crcv = 7) -> (Snode'=ID)&(Crcv'=8)&(help'=1);
// receive active node + busy signal.
[Csnd](time = iTimeA)&(Crcv = 8) -> (Cmsg'=8);
// send ack
[rcvC](Cmsg = 8) -> (Crcv'=9);
// receive Permission to Send.

// data loop start
[Csnd](time = iTimeA)&(Crcv = 9) -> 0.9:(Cmsg'=9) + 0.1:(Cmsg'=10);
// Sending: Cmsg 9 = packet. Cmsg 10 = last packet.
[](Cmsg = 10) -> (EoT'=true)&(Cmsg'=11);
// last packet sent = end of transmission
[rcvC](Cmsg = 9) -> (Cmsg'=0);
// ack for packet.
[rcvC](Cmsg = 11) -> (Crcv'=11);
// ack for last packet.
//data loop end

[](Crcv = 11) -> (wait'=true)&(Snode'=5);
// client goes idle after the end of transmission.

[rcvC](wait = true)&(Snode = 5) -> (wait'=false)&(Crcv'=12);
// client received the free signal. All nodes go out of idle.
[Csnd](time = iTimeA)&(Crcv = 12) -> (Cmsg'=12);
// Send "I'm alive" message to master.

[](failS1 = 1) -> (Cmsg'=0)&(Crcv'=0);
endmodule

module
Client2=Client1[rcvC=rcvC2,Csnd=Csnd2,Cmsg=Cmsg2,Crcv=Crcv2,Cboot=Cboot2,wait=wait2,ID=ID2,iID=iID2,iTimeA=iTimeB,synch1=synch2]
endmodule
module
Client3=Client1[rcvC=rcvC3,Csnd=Csnd2,Cmsg=Cmsg3,Crcv=Crcv3,Cboot=Cboot3,wait=wait3,ID=ID3,iID=iID3,iTimeA=iTimeC,synch1=synch3]
endmodule
module
Client4=Client1[rcvC=rcvC4,Csnd=Csnd2,Cmsg=Cmsg4,Crcv=Crcv4,Cboot=Cboot4,wait=wait4,ID=ID4,iID=iID4,iTimeA=iTimeD,synch1=synch4]
endmodule
module
Client5=Client1[rcvC=rcvC5,Csnd=Csnd2,Cmsg=Cmsg5,Crcv=Crcv5,Cboot=Cboot5,wait=wait5,ID=ID5,iID=iID5,iTimeA=iTimeE,synch1=synch5]
endmodule


//From here, Modules that handle the flag creation.

module Tr_Flag_A // 'is transmitting' flag
Tr_A: bool init false;
[](Snode=iID) -> (Tr_A'=true);
[](Snode!=iID) -> (Tr_A'=false);
endmodule
```

```
module
Tr_Flag_B=Tr_Flag_A[Tr_A=Tr_B,iID=iID2]
endmodule
module
Tr_Flag_C=Tr_Flag_A[Tr_A=Tr_C,iID=iID3]
endmodule
module
Tr_Flag_D=Tr_Flag_A[Tr_A=Tr_D,iID=iID4]
endmodule
module
Tr_Flag_E=Tr_Flag_A[Tr_A=Tr_E,iID=iID5]
endmodule

module synch_A // 'is in synch' flag
synch_A: bool init false;
[](synch1=true)->(synch_A'=true);
endmodule

module
synch_B=synch_A[synch1=synch2, synch_A=synch_B]
endmodule
module
synch_C=synch_A[synch1=synch3, synch_A=synch_C]
endmodule
module
synch_D=synch_A[synch1=synch4, synch_A=synch_D]
endmodule
module
synch_E=synch_A[synch1=synch5, synch_A=synch_E]
endmodule

module time_A //time partition flag
time_A: bool init false;
[](time=iTimeA)->(time_A'=true);
[](time!=iTimeA)->(time_A'=false);
endmodule

module
time_B=time_A[time_A=time_B,iTimeA=iTimeB]
endmodule
module
time_C=time_A[time_A=time_C,iTimeA=iTimeC]
endmodule
module
time_D=time_A[time_A=time_D,iTimeA=iTimeD]
endmodule
module
time_E=time_A[time_A=time_E,iTimeA=iTimeE]
endmodule

module Tpa   // unique time partition flag
tpa:bool init false;
[](iTimeA!=iTimeB)&(iTimeA!=iTimeC)&(iTimeA!=iTimeD)&(iTimeA!=iTimeE)->(tpa'=true);
endmodule

module
Tpb=Tpa[tpa=tpb,iTimeA=iTimeB]
```

```
endmodule
module
Tpc=Tpa[tpa=tpc,iTimeA=iTimeC]
endmodule
module
Tpd=Tpa[tpa=tpd,iTimeA=iTimeD]
endmodule
module
Tpe=Tpa[tpa=tpe,iTimeA=iTimeE]
endmodule

module SRA  // send , receive and ack flags
sndA:bool init false;
rcvA:bool init false;
[](Cmsg=9)|(Cmsg=10)->(sndA'=true);
[](Cmsg!=9)&(Cmsg!=10)->(sndA'=false);
[](Cmsg=0)|(Crcv=11)->(rcvA'=true);
[](Cmsg!=0)&(Cmsg!=11)->(rcvA'=false);
endmodule

module
SRB=SRA[sndA=sndB,rcvA=rcvB,Cmsg=Cmsg2,Crcv=Crcv2]
endmodule
module
SRC=SRA[sndA=sndC,rcvA=rcvC,Cmsg=Cmsg3,Crcv=Crcv3]
endmodule
module
SRD=SRA[sndA=sndD,rcvA=rcvD,Cmsg=Cmsg4,Crcv=Crcv4]
endmodule
module
SRE=SRA[sndA=sndE,rcvA=rcvE,Cmsg=Cmsg5,Crcv=Crcv5]
endmodule
```

Bibliography

[1] TTE Specification, TTA Group, 2008, available by request, at http://www.ttagroup.org

[2] H. Kopetz, A. Ademaj, P. Grillinger, K. Steinhammer. The time-triggered ethernet (tte) design. In Proc. of 8th IEEE International Symposium on Object-oriented Realtime Distributed Computing (ISORC), Seattle, Washington, May 2005

[3] W. Steiner, G. Bauer. Ethernet for space applications: TTE. In European Space Agency ESTEC, Noordwijk, The Netherlands, 2008.

[4] W. Steiner, B. Dutertre. SMT-based formal verification of a tte synchronization function. In Proceedings of the 15th international conference on Formal methods for industrial critical systems (FMICS'10), 2010.

[5] M. L. Puterman. Markov decision processes. John Wiley and Sons, INC, 1994.

[6] P. Grillinger, A. Ademaj, K. Steinhammer, H. Kopetz. Software implementation of a time-triggered ethernet controller. In Proc. of IEEE International Workshop on Factory Communication Systems, 2006.

[7] K. Steinhammer, P. Grillinger, A. Ademaj, H. Kopetz. A time-triggered ethernet (TTE) switch. In Design, Automation and Test in Europe, 2006.

[8] B. Dutertre, L. de Moura. The YICES SMT Solver. Available online at http://yices.csl.sri.com/tool-paper.pdf

[9] M. Jurdzinski, F. Laroussinie, J. Sproston. Model checking probabilistic timed automata with one or two clocks. In Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'07), 2007.

[10] PRISM Probabilistic Model Checker , University of Birmingham. Available online at http://www.prismmodelchecker.org/.

[11] P. Perny, O. Spanjaard, P. Weng. Algebraic Markov decision processes. In Proceedings of the 19th international joint conference on Artificial intelligence (IJCAI'05), 2005.

[12] M. Kwiatkowska, G. Norman, D. Parker. Symmetry reduction for probabilistic model checking. In Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems (QEST'09), 2009.

[13] T. Mhne, A. Vachoux, E. Villar. Proposal for a bond graph based model of computation in SystemC-AMS; In Proceedings of the Tenth International Forum on Specification and Design Languages (FDL'07), Barcelona, ECSI, 2007.

[14] M. Kwiatkowska, G. Norman, D. Parker. PRISM 4.0:Verication of Probabilistic Real-time Systems. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), 2001.

[15] A. Ademaj, K. Steinhammer, P. Grillinger, H. Kopetz, A. Hanzlik. Fault-Tolerant Time-Triggered Ethernet Configuration with Star Topology. In Proceedings of the 19th International Conference on Architecture of Computing systems (ARCS'06), Dependability and Fault Tolerance Workshop, 2006.

[16] OMG. Smart Transducer Specification TTP/A. Object Management group, 2002

[17] R. Alur and T. A. Henzinger. Reactive Modules. In Formal Methods in System Design 15:7-48, 1999.

[18] M. Kwiatkowska, G. Norman and J. Sproston. PCTL model checking of symbolic probabilistic systems. Technical report CSR-03-2, University of Birmingham, School of Computer Science. April 2003.

[19] B. P. Gerkey , S. Thrun, G. Gordon. Parallel stochastic hill-climbing with small teams. In Multi-Robot Systems: From Swarms to Intelligent Automata, Volume III, Netherlands, 2005.

[20] L. Zhang, Y.J. Cheng and X. Zhou. Rate avalanche: Effects on the performance of multi-rate 802.11 wireless networks. Simulat, 2009 Model. Pract. Theor., 17: 487-503. DOI: 10.1016/j.simpat.2008.09.003

[21] R. Williams. Computer Systems Architecture. Addison-Wesley, ISBN: 0201648598. 2001.

[22] P. Hazucha and C. Svensson. Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate. In IEEE Transactions on Nuclear Science, Vol. 47, No. 6, pages2586-2594, Dec. 2000.

[23] S. Schach. Object-Oriented and Classical Software Engineering, Seventh Edition. McGraw-Hill. ISBN 0-073-19126-4. 2006.

[24] F. Corella, M. Langevin, E. Cerny, Z. Zhou and X. Song State enumeration with abstract descriptions of state machines. In Proc. IFIP WG 10.5, 1995.

[25] Jonathan P. Bowen, Victoria Stavridou. Safety-Critical Systems, Formal Methods and Standards. Software Engineering Journal, 1993.

[26] M. Newborn. Automated Theorem Proving: Theory and Practice. First Edition, Springer. ISBN 978-0-387-95075-4. 2000

[27] S. Merz. Model Checking: A Tutorial Overview. In Proceeding MOVEP '00 Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes. Springer-Verlag, London, UK, 2001.

[28] W. Fokkink, M. T. Dashti, A. Wijs. Partial Order Reduction for Branching Security Protocols. In 10th International Conference on Application of Concurrency to System Design (ACSD), Braga, 2010.

[29] O. M. Fasan, J. Sanders, I. Rewitzky. How Model Checking Works. African Institute for Mathematical Sciences (AIMS), May, 2008.

[30] L. Lamport. The Temporal Logic of Actions. In ACM Transactions on Programming Languages and Systems, 16(3):872-923, 1994.

[31] Z. Manna, A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.

[32] E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems, 1986.

[33] M. Kwiatkowska, G. Norman, D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In International Journal on Software Tools for Technology Transfer (STTT), 2002.

[34] S. P. Meyn, R. L. Tweedie. Markov Chains and Stochastic Stability. London: Springer-Verlag, 1993. ISBN 0-387-19832-6. Second edition to appear, Cambridge University Press, 2009.

[35] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker. Automated Verification Techniques for Probabilistic Systems. In M. Bernardo and V. Issarny (editors) Formal Methods for Eternal Networked Software Systems (SFM'11), volume 6659 of LNCS, pages 53-113, Springer. June 2011.

[36] D. P. Bertsekas, J. N. Tsitsiklis. An analysis of stochastic shortest path problems, Math. Oper. Res., 16:580-595, August 1991.

[37] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker. Automated Verification Techniques for Probabilistic Systems. In M. Bernardo and V. Issarny, editors, Formal Methods for Eternal Networked Software Systems (SFM'11), LNCS. Springer, 2011. To appear.

[38] G. J. Myers. The art of software testing. New York, Wiley, c1979. ISBN: 0471043281, xi, 177 p. : ill. ; 24 cm.

[39] W. Visser. Software Model Checking. Research Institute for Advanced Computer Science NASA Ames Research Center. [online] Accessed on March 10th, 2011. http://visserhome.co.za/willem/presentations/ASE2002TutSoftwareMC-fonts.ppt

[40] V. Shmatikov. Probabilistic Model Checking for Security Protocols. [online] Accessed on March 10th, 2011. http://www.stanford.edu/class/cs259/WWW04/lectures/07-Probabilistic%20Model%20Checking.pdf