TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Madhushree Singh, 166807IVSM

PROVABLY CORRECT TEST DEVELOPMENT FOR TUT MEKTORY NANOSATELLITE SOFTWARE

Master's thesis

Supervisor: Jüri Vain

PhD

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Madhushree Singh, 166807IVSM

TTÜ MEKTORY NANOSATELLIIDI TARKVARA TESTIDE TÕESTATAVALT KORREKTNE ARENDUS

Magistritöö

Juhendaja: Jüri Vain

PhD

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Madhushree Singh

04.05.2018

1 Abstract

The aim of this thesis is to apply and evaluate the usability of Provably Correct Testing (PCT) workflow on the basis of TUT Mektory nanosatellite software testing case study. TUT Mektory Nanosatellite is a project being developed by students and professors with contributions from various industries and universities internationally.

To ensure the stability and reliability of the nanosatellite and the software associated with it, there have to be proper tests conducted on it before and after satellite launch. Testing can be conducted by different tools and methods, however, in this thesis, we will use Uppaal tool family and test execution environment DTron. Uppaal is an integrated tool used for modelling, verification, and testing of real-time systems. The usability analysis of the PCT process and tools incorporates estimates of time and computational resources spent on implementing the phases of the PCT workflow. In the course of process implementation, the abstract tests written using Uppaal modeling language and test interfaces have been implemented in Selenium which is used as an adaptor to execute the tests against the System Under Test (TUT Mektory nanosatellite software).

The results are validated and based on practical measurements of software quality metrics applied in the Mektory nanosatellite software development process. The correctness of test development steps is proven by Uppaal model checker against verification conditions extracted from the satellite design requirements.

This thesis is written in English and is 68 pages long, including 7 chapters, 30 figures, and 7 tables.

2 Annotatsioon

Käesoleva magistritöö eesmärgiks on rakendada ja hinnata tõestatavalt korrektse testimise (PCT) töövoo kasutatavust TTÜ Mektory nanosatelliidi tarkvara testimise juhtumianalüüsi põhjal. TTÜ Mektory Nanosatelliit on projekt, mida arendavad üliõpilased ja professorid koostöös erinevate tööstusharude ja ülikoolidega.

Et tagada nanosatelliidi ja selle tarkvara stabiilsus ja usaldusväärsus, tuleb seda põhjalikult testida enne ja pärast satelliidi orbiidile viimist. Testimine võib toimuda erinevate tööriistade ja meetoditega. Käesolevas väitekirjas kasutame Uppaali tööriistade perekonda ja testide täitmiskeskkonda DTron. Uppaal on integreeritud tööriist, mida kasutatakse reaalaja süsteemide modelleerimiseks, kontrollimiseks ja testimiseks. PCT protsessi ja tööriistade kasutatavusanalüüs sisaldab PCT töövoo etappide rakendamiseks kuluva aja ja arvutusressursside hinnanguid. Protsessi rakendamisel teisendatakse Uppaali modelleerimiskeeles esitatud testimudelite abstraktsed sisend- ja väljundsümbolid testi adapterite abil testitava tarkvara konkreetseteks sisenditeks ja konkreetsed tarkvara väljundid tagasi mudelil interpreteeritavateks sümbolväljunditeks. Töös demonstreeritakse, et mudeli ja mudelkontrolli päringu kitsenduste abil esitatud testi eesmärgid võimaldavad genereerida täidetavaid testijadasid, mis on kas testijooksu aja või testijada pikkuse mõttes optimaalsed.

Tulemused on valideeritud ja põhinevad Mektory nanosatelliidi tarkvara arendamise protsessis rakendatud tarkvara kvaliteedi mõõdikute praktilistel mõõtmistel. Katsetamisetappide õigsust tõendab Uppaali abil teostatud korrektsusomaduste verifitseerimine, mis näitab, et satelliidi juhtimistarkvara disain vastavab disaini projetis esitatud nõuetele.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 68 leheküljel, 7 peatükki, 30 joonist, 7 tabelit.

3 List of abbreviations and terms

AX.25	*Amateur X.25*
DPI	*Dots per inch*
DTRON	*Distributed Testing Realtime systems Online*
ESA	*European Space Agency*
FBD	*Function Block Diagram*
IUT	*Implementation Under Test*
MBT	*Model-Based Testing*
PCT	*Provably Correct Testing*
PCTD	*Provably Correct Test Development*
RPT	*Reactive Planning Tester*
SUT	*System Under Test*
TCTL	*Timed Computation Tree Logic*
TUT	*Tallinn University of Technology*
UI	*Unnumbered Information*
UPTA	*Uppaal Timed Automata*

4 Table of contents

1 Introduction 10
1.1 Motivation 11
1.2 Background 12
1.3 Problem Statement 12
1.4 Related Work 15
1.5 Research Questions 16
2 Preliminaries 18
2.1 Model-Based Testing 18
2.2 Provably Correct Test Development Workflow 19
2.3 Uppaal Timed Automata 20
2.4 TCTL Query language 22
2.5 Toolchain for the Provably Correct Test Development 24
3 Case Study 33
3.1 Description of Satellite 33
3.2 Satellite Architecture 34
3.3 System Under Test 36
4 Application of the Methodology and Tool Chain 37
4.1 Class Diagram of the System Under Test 37
4.2 Satellite Communication Architecture 39
4.3 SUT Modelling using Uppaal 40
4.3.1 Correctness of the Model 43
4.3.2 Test Purpose Specification 44
5 Analysis and Validation of Results 53
5.1 Time Taken to Model the SUT 53
5.2 Time Taken to Update Model for Test Purpose Specification 54
5.3 Timed Trace Analysis 55
5.4 Test Generation Time and Memory Usage Analysis 62
5.5 Summary of Analysis and Validation of Results 63
6 Conclusion and Future Work 64
7 Acknowledgment 65
References 66

5 List of figures

Figure 1: Online Model-based Testing workbench Uppaal Tron [11] 13
Figure 2: Provably Correct MBT Workflow [26] 20
Figure 3: A timed automaton modeling a simple lamp [27] 21
Figure 4: Examples of the query in Uppaal [26] 23
Figure 5: Uppaal Tool Overview... 24
Figure 6: Uppaal ‘Help’ menu... 25
Figure 7: Simulator in Uppaal 25
Figure 8: Verifier in Uppaal 26
Figure 9: Adaptor class & Reporter class [22] 29
Figure 10: DTron API domain model [22]... 31
Figure 11: Distributed testing data-flow in DTron [22] 32
Figure 12: TUT Mektory Nanosatellite System [23]... 34
Figure 13: TTU Mektory Nanosatellite System Segments.. 35
Figure 14: TTU Mektory Nanosatellite Communication Protocol [23] 36
Figure 15: Class Diagram of Satellite Communication Protocol [24]........................... 37
Figure 16: Satellite Communication Protocol Architecture 39
Figure 17: Satellite Communication Protocol 40
Figure 18: Nanosatellite Model using Uppaal Tool 41
Figure 19: Test Purpose 1 Model 45
Figure 20: Test Purpose 1 Verification.. 45
Figure 21: Test Purpose 1 Witness Trace (Step by Step) 46
Figure 22: Test Purpose 2 Model 47
Figure 23: Test Purpose 2 Verification.. 47
Figure 24: Test Purpose 2 Witness Trace (Step by Step) 48
Figure 25: Test Purpose 3 Model 49
Figure 26: Test Purpose 3 Verification.. 49
Figure 27: Test Purpose 3 Witness Trace (Step by Step) 50
Figure 28: Test Purpose 4 Model 51
Figure 29: Test Purpose 4 Verification.. 52
Figure 30: Test Purpose 4 Witness Trace (Step by Step) 52

6 List of tables

Table 1: Time Taken for Modelling SUT 53
Table 2: Time Taken for Test Purpose Specification and Verification 54
Table 3: Timed Trace for Test Purpose Specification 1 55
Table 4: Timed Trace for Test Purpose Specification 2 55
Table 5: Timed Trace for Test Purpose Specification 3 57
Table 6: Timed Trace for Test Purpose Specification 4 59
Table 7: Time and Memory Usage Analysis 62

1 Introduction

TUT Mektory nanosatellite software project [1] is being developed by students and professors with contributions from various industries and universities internationally. The goal of the nanosatellite project is to help students acquire practical experience in space technology and get hands-on experience in utilizing their engineering skills. The construction of the satellite is from 2016 to 2018, and it has been planned to be launched in the year 2018. The main goal of the satellite is to establish an Earth station and to operate the satellite for various reasons with one of them being to keep track of environmental changes. This encourages not only the youth but everyone to have an interest in knowing and building newer technologies.

To assure the quality of any software, application or system, there must be proper tests conducted on it to proclaim its functional correctness, reliability and effectivity. Thus, to ensure that the TUT Mektory satellite is reliable and satisfies performance requirements, there are various tests conducted on the satellite and on the software with which it is associated.

The ground station software that interacts with the satellite is one of the main applications to be tested but other software components involved in full solution influence the overall features as well. Therefore, integration testing should address the issues of collaboration between all the components that contribute to the mission success.

1.1 Motivation

The first question that comes to one's mind is that, *"What steps are required to ensure that a satellite is correctly built and is ready to be launched?"*. The answer to this is that the satellite should be thoroughly tested for all critical aspects of its design. Before a satellite is launched, there must be corrective tests conducted on the satellite device and on all the software that controls it.

There are many ways in which the system can be tested, one of them addressing the testing productivity and automation issues the best is Model-based testing (MBT). Model-based testing can be defined as the automation of the design of black-box testing [2]. The main difference between manual black box testing and model-based testing is that in the former case the test scripts are written manually based on the requirements specified, whereas, in latter case we create a model of the expected System Under Test (SUT) behavior, thus capturing the requirements the test case is meant for. After this, the model-based testing tools are used to generate tests from that model automatically [2].

Uppaal Tron [3] is a tool which is developed for testing functional, performance and timing correctness. For the distributed systems, the tool called DTron [4] which stands for *"Distributed Testing Realtime systems ONline,"* is used and it is based on the Uppaal model checker and the Uppaal Tron tool.

The problem in testing is that it is hard to determine which tool is the best for given test purpose, and whether the results developed from the testing tool can be proved to be correct. In this thesis, the validation will be based on practical measurements of software quality metrics applied in the Mektory nanosatellite software development process. Uppaal tool family [5] is an integrated environment which enables users to validate, verify, and model real-time systems. The correctness of test development steps will be proved by Uppaal model checker.

The idea of this thesis has merged from the real engineering problem. The TUT Mektory nanosatellite is being actively developed, and it has been finalized to launch it in the year 2018. So, this work is based on genuine need and interest as a pilot project to be scaled up in later collaborative projects with ESA.

1.2 Background

There are different types of test automation tools such as Torx, Watir, and others which are in use in software testing practice. Majority of available tools support functional testing. So, instead of using many narrowly specialized tools, Uppaal Tron differs from those being a model-based testing tool and is developed for testing functional, performance and timing correctness all combined. Uppaal tool being an integrated environment enables users to validate, verify, and model real-time systems.

Selenium is also one great framework to write test cases and to implement them. Automation testing improves the quality of testing and reduces manual effort [14], and Selenium is an open source tool which is very popular for testing web applications [14]. Selenium along with TestNG framework overcomes many limitations of the JUnit framework [15]. TestNG framework covers all unit, functional and integration testing [15].

In model-based testing, the model formally captures requirements of the *System Under Test*. This model can be used to generate test cases. The author's choice is Uppaal Timed Automata (Uppaal TA) because the timed behavior of the state transition can be expressed and Uppaal tool family being an integrated environment enables users to validate, verify, and model real-time systems.

1.3 Problem Statement

Satellites are used for many different purposes. Some common ones are Earth observation satellites for military usage, weather monitoring satellites, communication satellites, navigation satellites, etc.

Over the last few years, there have been several satellites that have been launched successfully. However, there have been many reported cases of failed launches of satellites, space shuttles, and rockets from across the globe. These failures cause a significant impact on the time and money invested in the making of the satellites. Usually, the expenses are in millions and even billions of dollars. Therefore, there must be proper maintenance and regular quality check-ups done to ensure good performance and stability before and after the launch of the satellite.

To ensure good performance and stability of the nanosatellite, there have to be proper tests conducted on it and on the system, which operates the satellite through the Earth station.

The system can be tested by different tools and with different software testing approaches. Model-Based Testing (MBT) approach is an approach which improves the test coverage as it addresses the project resources and time thus resulting in good test quality [6]. The reason for using MBT is that it has many benefits over the classical testing approaches and these benefits are specified below [7].

The scope of this thesis will include usage of following MBT tools and related to them test automation approaches:

- Uppaal and Uppaal Tron – For modeling, verification and online testing of real-time systems [8] [9].
- DTron – For testing in the distributed systems such as the ground system which operated the nanosatellite [4].
- Selenium – It will be used as an adaptor to implement the tests with the help of Java as a programming language [10].
- Uppaal model checker – Proving the correctness of the test developed [7].

The configuration of MBT tools and their interconnections are depicted in Figure 1 [11]:

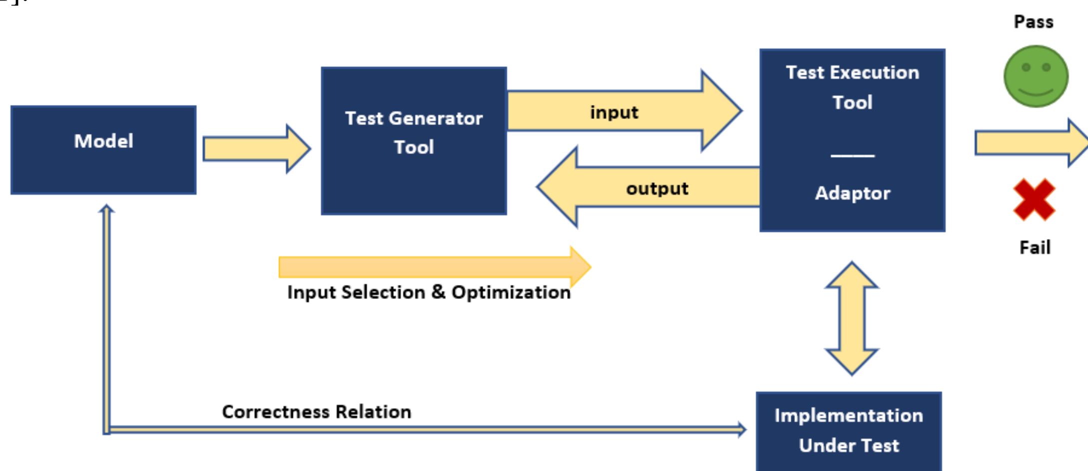

Figure 1: Online Model-based Testing workbench Uppaal Tron [11]

According to Figure 1, the Model will be developed from the requirements specification of SUT and constrained by the test purpose given in test specification. The tests are generated either online by Tron tool during test execution or offline from the witness traces provided by the Uppaal model checker. The properties the model checker is verifying originate from the test purpose specification and are specified as TCTL formulas. Since Uppaal supports not full TCTL, some of the properties need to be specified as SUT model constraints. The technical details of test case model construction will be explained in Section 4 (Application of Methodology and Tool Chain). After test model and test purpose are specified their correctness and tests feasibility are verified using Uppaal model checker. Generated test are executed in Uppaal Tron or its extension for distributed systems DTron. Additionally, Selenium can be used as an adaptor to communicate between model executing environment and the Satellite web interface. Adaptor transforms the symbolic test inputs/outputs specified in the model to executable by SUT inputs and outputs.

The test development workflow comprises steps such as SUT modeling, test purpose specification, test generation, deployment, and execution. The thesis has to apply all these steps to satellite ground station software:

- The first task is to formalize the SUT software specification using Uppaal TA.
- The second task is extracting and formalizing the correctness criteria for the outcome of each testing process step. This involves both general correctness properties outlined in [16] as well as application specific properties unique to given software.
- The third task is to prove the correctness of the steps by model checking using Uppaal model checker or Divine tool.
- The fourth task is to validate the approach by comparing the performance of the used MBT process and the number of detected faults with those reported by control group that does manual testing.

1.4 Related Work

In the past few years, there has been some research work done in the field of provably correct test development for various systems. One of the research work is closely related to the current thesis, but it is based on *Provably correct test development for timed systems* [12]. In the paper, there is a study done on the various ways by which the correctness of test derivation steps can be ensured and how the test results can be made trustable by the testing process [12].

The paper also focusses on model-based testing of software systems with timed constraints [12]. Model-based testing process is considered consisting of following steps:

- The first step is to model the System Under Test (SUT) which is also referred as Implementation-Under-Test (IUT).
- The second step is where the test purpose gets specified.
- The third step is where the tests get generated.
- The final step is where the generated tests get executed against the System Under Test(SUT).

In paper [12], the authors use Uppaal Timed Automata (Uppaal TA) as a formalism to model the SUT behavior. The use of Uppaal TA is because of the need to test the SUT with timing constraints so that the propagation delays between the tester and the SUT is taken into account.

Another paper [13] is based on the model-based test suite generation offline using the Uppaal Model checker for the Function Block Diagrams. The main idea of the paper is threefold [13]:

- The authors propose a transformation of Function Block Diagram (FBD) into timed automata models.
- The generation of tests is using Uppaal Model Checker.
- The applicability of the author's method is illustrated on the railway industry for management of trains and control software.

Remote testing explained in the paper [28] gives a vivid overview of MBT and conformance testing. In conformance testing, the *SUT* is considered as a black box where only the input and output are controllable and observable respectively [28]. When testing, test cases are executed against the *SUT* and a decision is made on the test case which is either a pass or fail or inconclusive. This decision shows the input-output conformance relation (IOCO) between the test specification and the *SUT*. Uppaal Timed Automata (Uppaal TA) is then used for modeling the *SUT* behavior and DTron so as to have distributed execution.

A somewhat similar approach has been used in the paper [29] where the authors applied model-based testing to improve the state of art of integration testing. The paper provides an automated approach for generating the model for a robot case. Uppaal Tron is used as the test engine and DTron as the adaptor generating framework [29].

A detailed explanation of Provably Correct Test Development Workflow and the use of Uppaal Tron and DTron is elaborated in the paper [30]. The author's research is based on an extensive study and survey of model-based testing emphasizing on models which are verified, and to enable provably correct on-line testing which was applied to the case study on a street light control software testing.

However, in this thesis, stress is put on the correctness of the test development steps and on the specifics of the SUT - Mektory nanosatellite communication protocol. The author will focus on the correctness proving using Uppaal model checker [8] and on offline test generation using verification traces.

1.5 Research Questions

TUT Mektory Nanosatellite being one of the current projects at the University requires trustable testing results to ensure its successful launch and good performance even after its launch.

With the above-mentioned motivation, the main RQ for this thesis is, "*How can the tests and testing results be proved to be correct?*" The nanosatellite not only requires tests to be conducted on it, but one also has to ensure that the tests conducted are correct and validated by some rigorous method.

Model-based testing will help to achieve the goals of the thesis in formally verifiable way.

This leads to the following concrete research questions:

RQ1: "*If testing is based on Model-Based Testing then what are the correctness criteria which can be applied to test cases?*". Our hypothesis is that in addition to generic correctness criteria such as deadlock freedom, input enabledness, connectedness, coverage correctness, etc. there are SUT and test case specific correctness criteria that need to be taken into account when concrete test sequences are generated.

RQ2: *"How can the test purpose be specified in case of Model-Based testing?"* – Our hypothesis is that the Uppaal TA modeling language is expressive enough for stating the test purposes and test model constraints formally. Due to the formal semantics the Uppaal model checker can be applied for verifying the correctness criteria of created test models.

RQ3: "*What is the relevance of Model-based testing for the TUT Mektory nanosatellite?*" – The relevance of Model-Based testing for TUT Mektory nanosatellite will be explained with analysis of timing constraints and time taken to verify test purposes, thus explaining why Model-Based testing is the best choice.

2 Preliminaries

A short introduction and explanation of the tools and testing methods used in this thesis will be given in this section.

2.1 Model-Based Testing

Model-based testing can be defined as the automation of the design of black-box testing [2]. The main difference between manual black box testing and model-based testing is that in the former case the test scripts are written manually based on the requirements specified, whereas, in latter case we create a model of the expected System Under Test (SUT) behavior, thus capturing the requirements the test case is meant for. After this, the model-based testing tools are used to generate tests from that model automatically [2].

Model-based testing has four main approaches [2]:

- The test input data is generated from a domain model
- The test cases are generated from an environment model
- The test cases with oracles are generated from a behavior model
- The test scripts are generated from abstract tests

The first approach is where the domain of the input values is the model, and the test generation is a selection of the subset of those values to obtain the test input data [2].

The second approach is based on environment model, and it describes the environment of the System Under Test. Usually, in this approach, it is not possible to determine the output values because this approach does not model the behavior of the SUT. Therefore, it is difficult to determine if the test failed or passed.

The third approach is the generation of test cases with oracles. Oracles here determine the expected results of the SUT or some automated check on the results to see if it is correct.

This approach is rather difficult because the output values generated are checked for their correctness. In this approach, the test generator should know about the behavior of the SUT so that it can check or predict the output values [2]. The greatest advantage of this approach is that it is the only approach out of the four which focusses on the whole design problem from selecting the input values to generating sequences of operation calls to generating the executable test cases.

The fourth approach is different as it assumes that an abstract description of the test case is given. This abstract description can be in the form of a UML sequence diagram or a high-level procedural call. The main focus of this approach is to transform the abstract test case into an executable low-level test script.

Model-based testing is more like automation of black-box test design. The model should be abstract, concise and expressive enough to represent behaviours of the SUT: concise so that it is easy to validate and is not too long to write and precise enough to explain the behavior that is to be tested [2]. The test cases can usually be automatically generated from the model with the help of a model-based testing tool.

2.2 Provably Correct Test Development Workflow

Provably Correct Test Development (PCTD) method solves the issues of trustability and provides great conclusiveness for automatically generated tests and procedures. This improves the productivity and quality of entire software development process. In many commercial testing tools, it is left to the user responsibility to ensure that the test results achieved through automation are trustable. That leaves it open if the testing results are conclusive unless there are some exhaustive checks for its correctness. PCTD helps to solve this issue with ease [12].

Provably Correct Test Development method involves Model-Based Testing, with the focus of two main study subjects: Test Automation and Test Correctness. The following are the three main steps involved in Model-Based Testing [12]:

- Modeling a System Under Test
- Specifying the test purpose

- Generating the test
- Executing against the system under test

The Figure 2 shows a tool supported workflow suggested in [26] on how a model-based test is developed and proved to be correct.

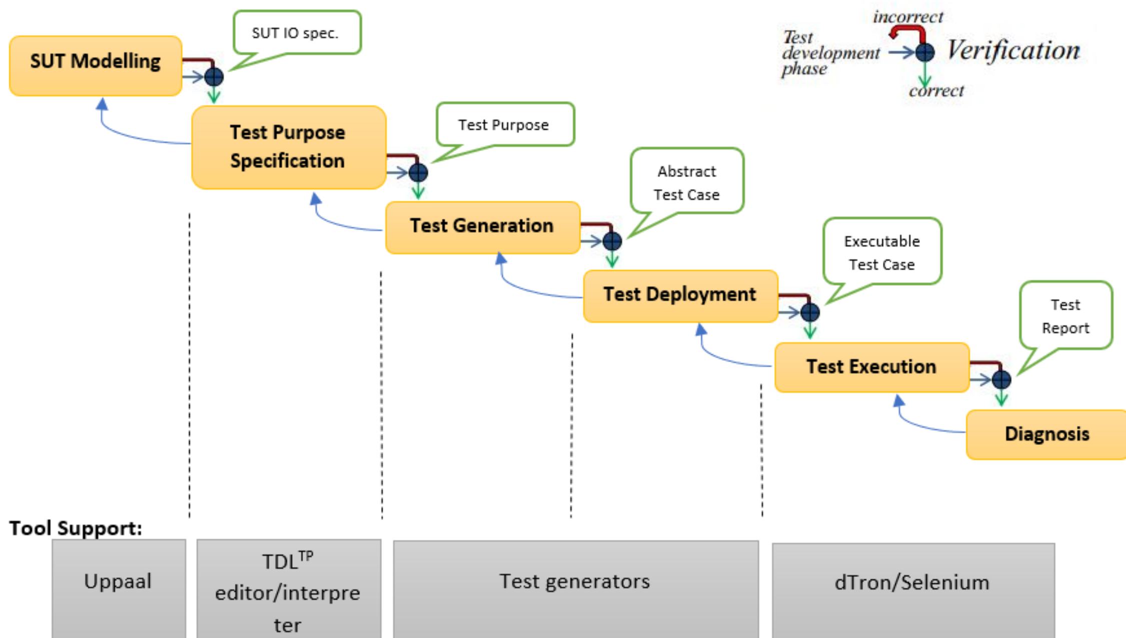

Figure 2: Provably Correct MBT Workflow [26]

2.3 Uppaal Timed Automata

Uppaal is a tool which is used for modelling, simulation and verification of real-timed systems [3]. Uppaal modelling language is based on extension of timed automata introduced by Alur and Dill. Original timed-automata have only clock variables and there are finite number of control states or so-called locations [3].

In Uppaal, the model is a network of several timed automata composed in parallel [3]. In addition to clocks the model is elaborated with data variables that are part of the state. These variables of Boolean and integer type are like the ones in programming languages which can be used for calculations and different operations. The state of a system is

defined by the clock constraints, values of the variables and the control locations of all the automata. Every automaton may fire a transition which leads to a new state.

One timed automaton modeling example of a simple lamp is shown below:

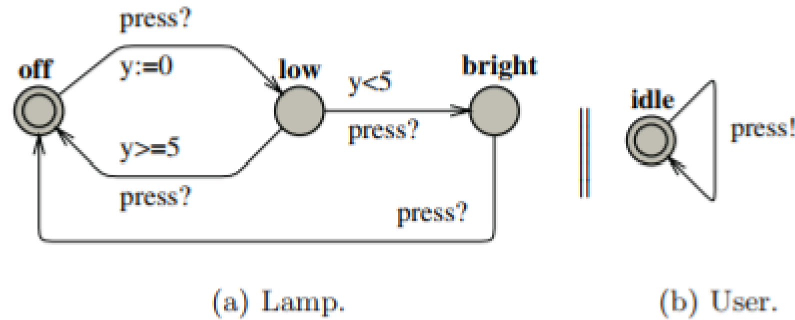

Figure 3: A timed automaton modeling a simple lamp [27]

We see from the Figure 3 that there are two automata of different roles. One for the User and the other is the Lamp itself. There are three states of the lamp: off, low and bright. The change of states takes place with the way the user presses the lamp button. If the user presses the button for the first time and once then the lamp is turned on. If he presses it again, then the lamp is turned off. But, if the user presses the button twice and fast, then the lamp is turned on and becomes bright. The clock 'y' of the map is used to detect if the user was fast and pressed the button two times within 5 seconds ($y<5$) or slow ($y>=5$) [27].

In more technical terms, Uppaal Timed Automata can be defined as a closed network of timed automata which can be called as processes [17]. Locations are the nodes in the automata graph and edges are the directed vertices between the locations [18]. Automaton state consists of the current control location and the valuation of all clock and data variables. Channels are the synchronization links between the processes. Channels having unique id-s, e.g. 'ch' relate a pair of transitions in parallel processes [18]. Here, the synchronized edges are labeled with symbols for input and output actions denoted by 'ch?' and 'ch!'.

Mathematically, let us assume that $\sum$ is a finite alphabet of actions (a, b, c, …), and C is a finite set of real-valued variables (x, y, z) denoting clocks [18]. A conjunctive formula

of atomic constraints of the form $x \sim n$ for $c \in C$, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in N^{+}$. We use G(C) to denote the set of clock guards. A timed automaton 'A' can be defined as a tuple (L, l_0, E, Inv) in which L is the finite set of locations, $l_0 \in L$, and is the initial location, E is the set of edges where $E \in L \times G(C) \times \Sigma \times 2C \times L$, and Inv is where $L \rightarrow I(C)$ assigning the invariants to locations where we restrict the constraints in the form of $x \leq n$ or $x < n$, $n \in N^{+}$.

2.4 TCTL Query language

The query language used in Uppaal model checker is TCTL (Timed Computation Tree Logic). It consists of path formula and state formulas [3]. State formulas express the properties of individual states whereas path formulas express the paths properties of the model.

State formula. An example of a state formula is an expression 'i == 7', which implies that the expression is true in the state where 'i' is equal to 7. In Uppaal, if there is a deadlock (no outgoing transitions from the state itself or any of its successors [3], then the syntax for it is the keyword 'deadlock' itself.

Path formula. The path formulas are further classified into three types:

- Reachability – Reachability property asks whether a given state formula, 'φ', is satisfied by any reachable state. In other words, starting at the initial state is there any existing path such that φ is satisfied along the path. The syntax used in Uppaal is E <> φ. One such example is when a communication protocol is modeled, and it consists of a sender and a receiver. So, in this protocol, it becomes essential to know if the sender is able to send messages and if the receiver actually receives the message or not [3].

- Safety – Safety property is where something bad is never going to happen or more of like being on a safe side. For example, in any game, there arise many situations where a player is in a situation where there are chances of winning the game and that the player will not lose the game. Assuming 'φ' to be the state formula, assuming φ is true in all reachable states having the path formula A [] φ, such that

E [] φ states that there must be a minimal path in which φ is always true. So, the syntax in Uppaal is A [] φ and E [] φ [3].

- Liveness – Liveness is the property where something will happen. For example, when an air-conditioning is turned on by pressing the '*On*' button, then it gets turned on. In Uppaal, Liveness is expressed as A<> φ, which means that φ is ultimately satisfied [3].

Some examples can be seen from the following Figure 4:

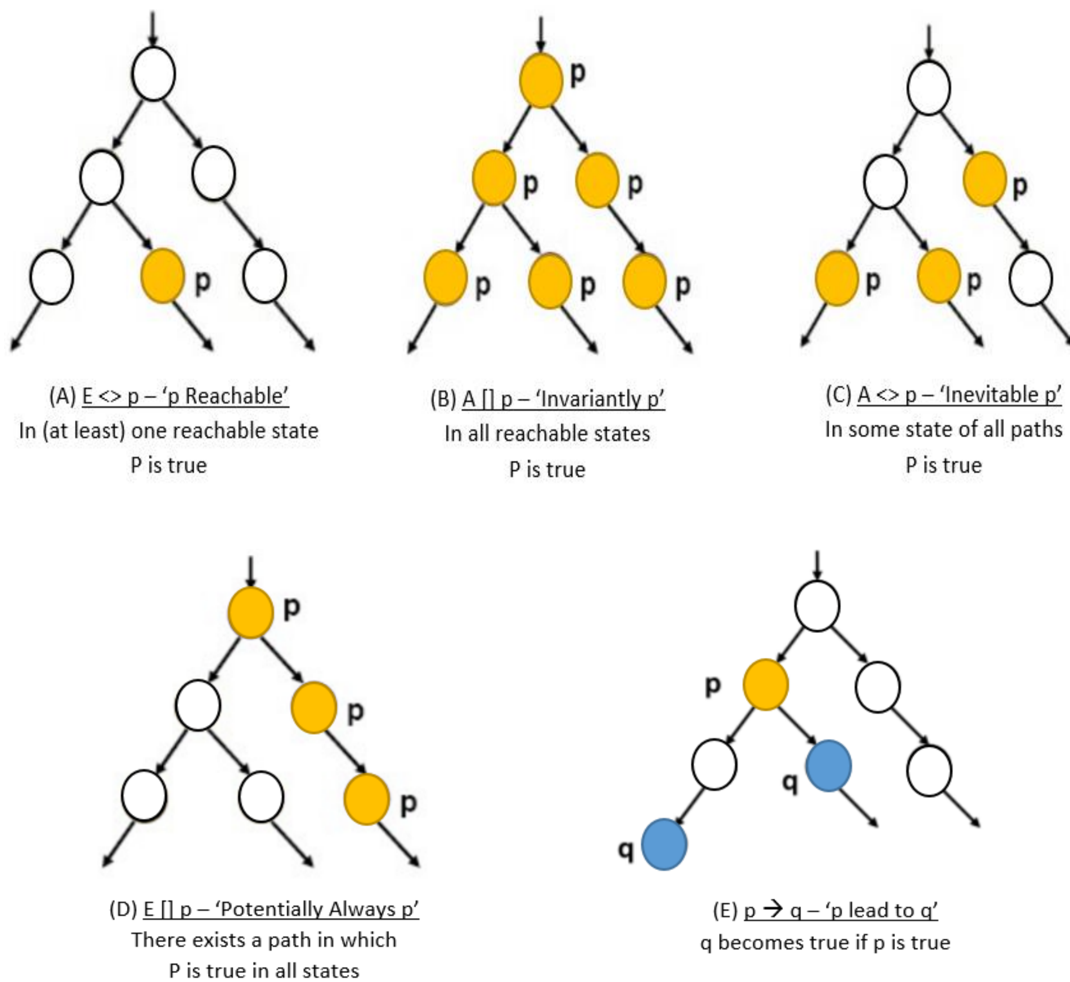

Figure 4: Examples of the query in Uppaal [26]

2.5 Toolchain for the Provably Correct Test Development

In this section we give an overview of main tools used in our PCTD process.

Modeling with Uppaal

Uppaal Tool has two main GUI components: Menu bar options and tabs (editor, simulator, and verifier) [19].

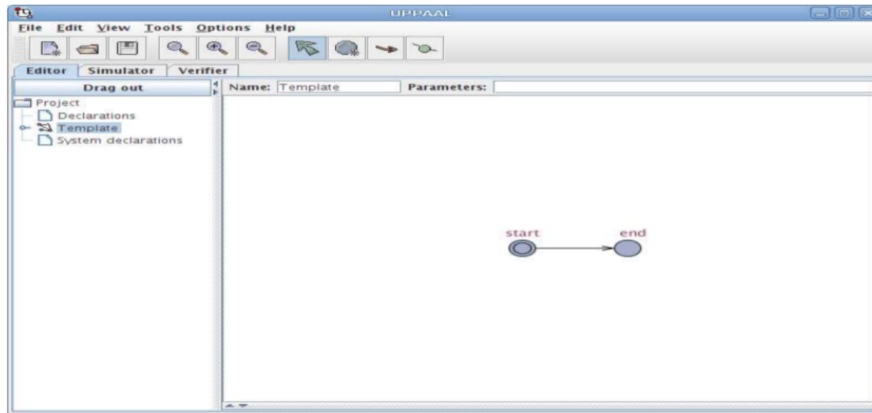

Figure 5: Uppaal Tool Overview

Figure 5 shows the Editor window which is the first tool in the process. When Uppaal starts, an initial location is already pre-created for us. The initial location is distinguished from other locations by two circles in it. To add another location 'Add Location' is clicked from the menu bar and then the location next to the initial location is used to place it. Similarly, 'Add Edge' button is used to connect the two locations and then 'Selection Tool' can be used to select the model element for its further specification. This way, we have the first automation ready.

The menu bar has essential features related to the modeling. 'Help' menu describes most of the functionalities of the menu bar and explains the syntax and the GUI in details.

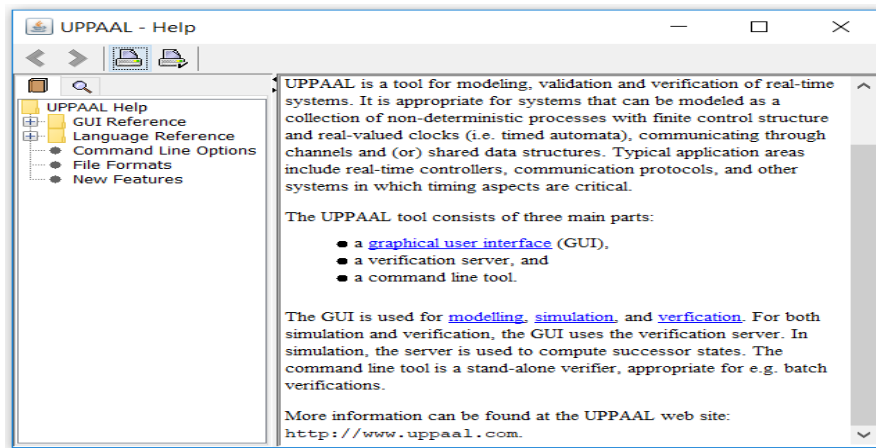

Figure 6: Uppaal 'Help' menu

Once the editing of the model has been done, one can click on the Simulator tab to get started with the simulator.

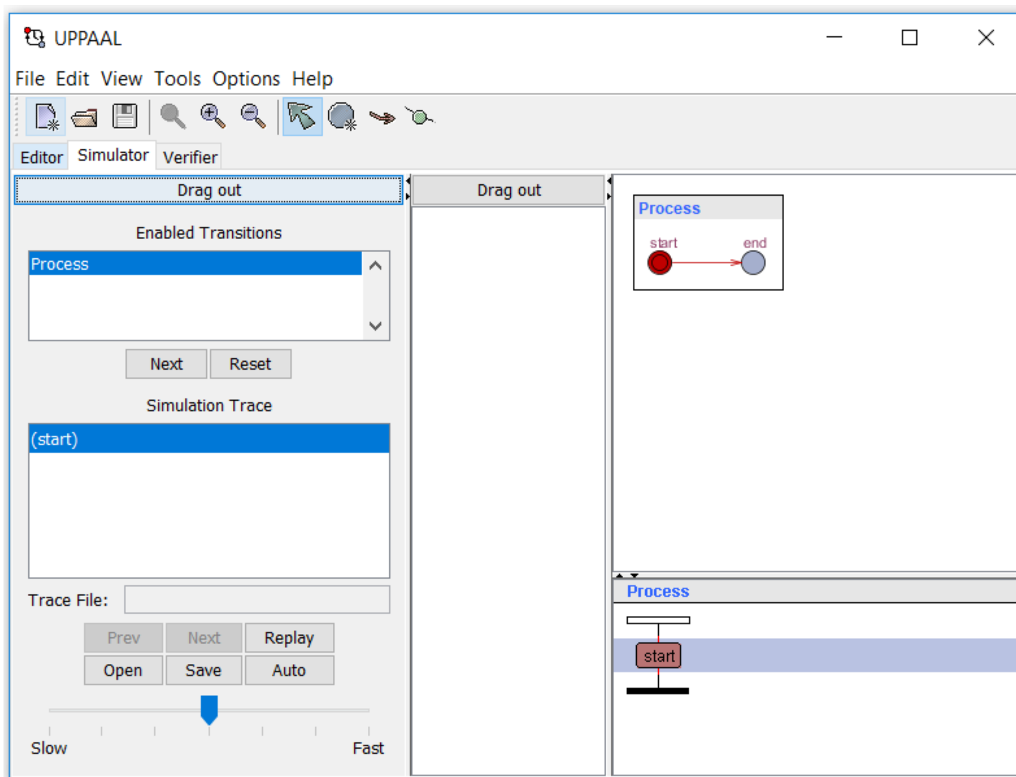

Figure 7: Simulator in Uppaal

The left panel in the Simulator tab is where most of the controls are. We can choose the transitions (Enabled Transitions) which would work on an existing trace (Simulation Trace). In the middle column (Drag out) the valuation of variables is shown, the process

model itself is located on the right-top, and the right-bottom shows the message sequence chart of the processes [19].

To simulate it we click on one of the Enabled Transitions and click on Next. The process at the right progresses one step and we see that the right-bottom view also has a change. This way, we have simulated the process and can proceed to the next step of Verification.

The Overview is where we can write the query and check for it satisfaction by clicking on the 'Check' button on the right. In figure 8, the query is 'E<>Process.end' means that we would like to know *if it is possible to reach the end location in the process* named *Process* we have created above. And after checking, we see that the property is satisfied.

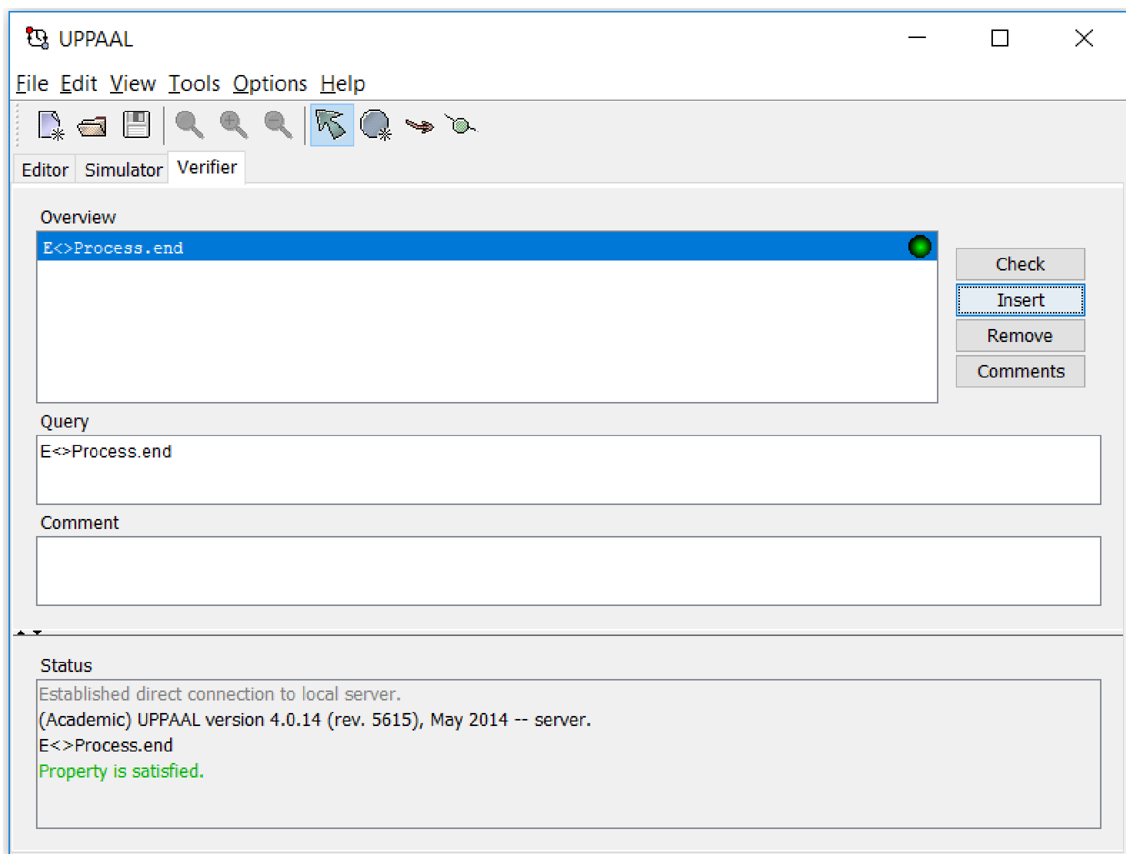

Figure 8: Verifier in Uppaal

a. **Test Purpose Description language - TDL**

Every organization with complex test cases needs convenient and compact notations for the test specifications. The Test description languages (TDLs) consists of two components: System Under Test (SUT) modeling language TDL^{SUT} and the test purpose specification language TDL^{TP} [20]. TDL^{SUT} captures the important features for SUT development. It is relatively independent of test cases. SUT specification language in this thesis approach is Uppal TA. The model elements referred in the model are called test coverage items and they form the set of ground level terms referred in more abstract test purpose definition language TDL^{TP}. TDL^{TP} expressions define the test scenario, it is more abstract than TDL^{SUT}, and its expressions must be interpretable on the test model.

Syntax of TDL^{TP}

A Boolean variable called *Trap* labels a test coverage item of the test model [20]. The traps are auxiliary Boolean variables' assignments that are added to the test model edges for tracking the test progress and for estimating the model coverage by test a run. The trap can also be thought of as a milestone indicating the progress of SUT model exploration. By default, the trap variable is set to *false*. It is set to *true* when the edge labeled within the trap is visited during test execution. An elementary trap is where the update function is unconditional, meaning, *tr: = true*. The general assumption is that the update functions are not recursive, the arguments are defined whenever the edge labeled with the trap is executed, and the trap names are unique. For brevity we call the set of traps with same name prefix a trap set. For enhancing the specification of trap set properties we represent a trap set also as a Boolean array each element of which is an individual trap.

Let, $Edge(M^{SUT}) \rightarrow$ Set of edges of SUT model M^{SUT} [20],

$e \rightarrow Edge(tr)$, function mapping tr to an edge $e \in Edge(M^{SUT})$ which is labeled with tr and C^{tr}, the updated constraint of tr.

So, for the test case TC, we say a set $\{ (e, C^{tr}) \mid e \in Edge(M^{SUT}) \}$ to trap the set TS^{TC} and assume that the edges e are decorated with updates $tr \leftarrow C^{tr}$ for all $tr \in TS$.

Given a trap set TS we distinguish two types of constraints - universal and existential - and denote them as ∀TS and ∃TS respectively. ∀TS means that all traps and ∃TS mean that at least one trap of the set TS must be set true for a test run to be PASSED. For shorthand, we use symbol ALL to denote a trap set including all elementary traps tr of M^{SUT} such that Edge(ALL) = Edge(M^{SUT}). Non-atomic TDL^{TP} expressions SE are defined as follows [20]:

- Universal and existential trap sets TS:

SE ::= ∀TS | ∃TS

- The complement of TS:

SE ::= !TS, !TS including elementary traps tr such that Edge(tr) ∈ Edge(ALL) and Edge(tr) ∉ Edge(TS).

- Logic connectives:

SE ::= not SE | SE_1 & SE_2 | SE_1 or SE_2 | SE_1 => SE_2

- "Next" operator ";":

SE ::= TS_1 ; TS_2 specifies a set of pairs of M^{SUT} edges (e_1, e_2), s.t. e_1 ∈ Edge(TS_1), e2 ∈ Edge(TS_2) and pre(e_2)=post(e_1). pre(e) denotes a pre-state of the edge e where e is enabled, and post(e) is post-state after executing e). If only one of the sets (either TS_1 or TS_2) is specified "_" is used instead of other, meaning "any".

- "Leads to" operator:

SE ::= SE_1 ~> SE_2

- Time-constrained "leads to" operator:

ˇ SE ::= SE_1 ~>[$®_{DL}$] SE_2, where ® ∈ $\{ <, =, >, \geq, \leq \}$ and DL is a deadline of event SE_2 w.r.t. event SE_1. DL ∈ N^+ and DL < TestTimeOut, where TestTimeOut is upper time bound to the test run if specified. We call DL an absolute deadline if the SE_1 is specified with wildcard symbol "_", i.e., the deadline is w.r.t. the start moment of the test run [20].

- Parameterized iteration: SE ::= #SE ® n where #SE denote the number of occurrences of SE, ® $\in$ { <, =, >, $\geq$, $\leq$} and n $\in N^+$.

b. Test Execution – TRON/DTRON

The functionality of Uppaal is extended with test execution tool TRON and its extension for distributed systems DTRON [22]. This section will give a brief explanation of the DTRON architecture and its functionality.

- **DTRON architecture**

  DTRON has been built over Uppaal model checker and Uppaal TRON. It has three important components:

  - *Adaptors/Reporters:*
  - Adaptors help in communication with instances of TRON tool. The symbolic input of the test model is converted by Adaptors to concrete executable test input which is then communicated over Spread toolkit and sent to SUT [22].

    Uppaal TRON has an API in C and Java for this interaction. The API has two classes: Reporter and Adaptor

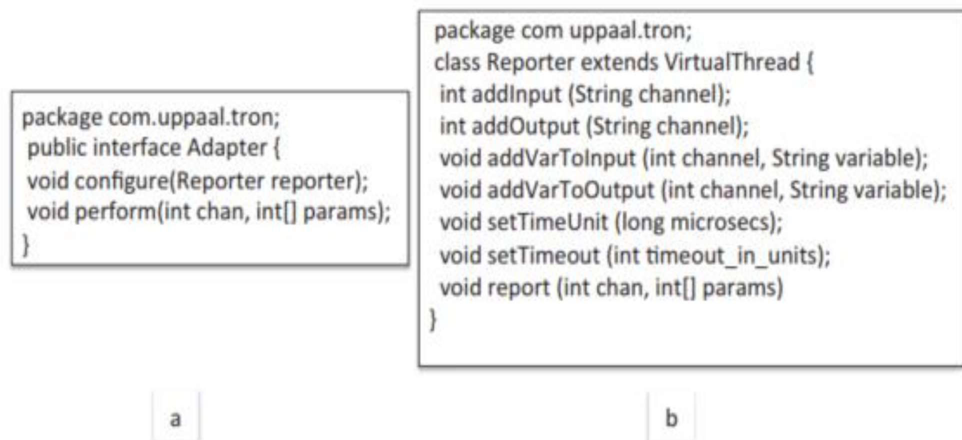

Figure 9: Adaptor class & Reporter class [22]

Reporter

Reporter handles the connection between the model and the Uppaal TRON runtime. There is a *'handshake'* phase established at 'void configure(Reporter) which is invoked by Uppaal TRON thus allowing access to the Reporter object for session configuration of [22]:

1. Time Unit – Determining the time for the unit of the model check (in microseconds).

2. Timeout – Time limit to the model execution.

3. Input/output channels – The channels are declared between the SUT and the Uppaal model. By naming convention, the input channels must have prefix 'i_' and output channels prefix 'o_'.

Adaptor

The main drawback of Uppaal TRON is that it does not support distributed testing. It can communicate with a single port at a time. DTron is the extension of TRON that enables test interaction over multiple ports simultaneously. An important component in DTron is Spread Toolkit.

- *Spread Toolkit* takes care of message serialization. Spread toolkit is used to interface the runtime with different platforms, Google Protocols, and languages. Mainly for message interchange Spread Toolkit is used by DTron [22]. The pattern for the messages is like *publish-subscribe*. Subscriptions are done by local testers that get messages and get notified whenever new messages are available by the callback.

  Spread provides many services such as point to point communication and group communication. It ensures that there will be no faults during the message exchange in local and wide area networks. Spread guarantees six levels of messages, namely: unreliable, reliable messages, FIFO – by the sender, casual, agreed and safe (total order) [22]. In DTRON, the messages

are flagged as safe which specifies that the SUT semantics is not broken and that they are produced in the same order to all the recipients [22].

- DTRON *API:* It has a direct connection with SUT ports, so DTron API acts as a local test adaptor. The API can also be used to write Adaptors which would interact with the SUT. The DTron API domain model can be seen in the following figure:

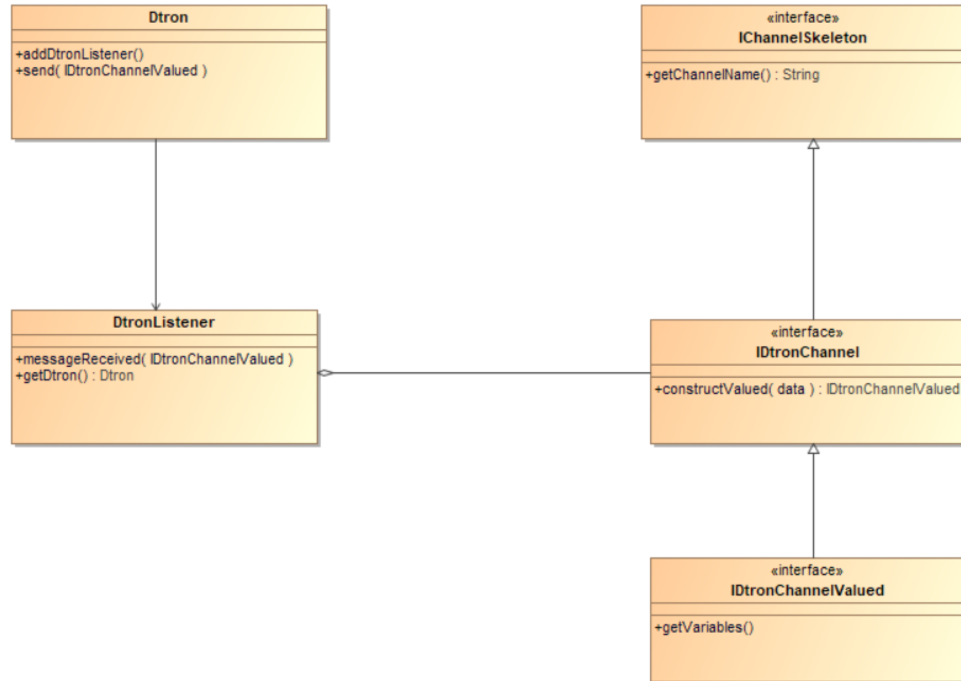

Figure 10: DTron API domain model [22]

The class DTron handles the connection to the Spread broker and allocates or releases resources. In this way, this is also the main entry point for the API. The DTron connection is used to assign *DtronListener* [22]. And, as we can see in the domain model, the listeners are based on the *IDtronChannel* which holds details about the model channels.

- *Distributed Execution:* When SUT has distributed architecture, it has usually multiple ports at different geographic locations for interaction between the tester and the SUT. The following diagram shows the conceptual view of distributed runtime deployment configuration of DTRON.

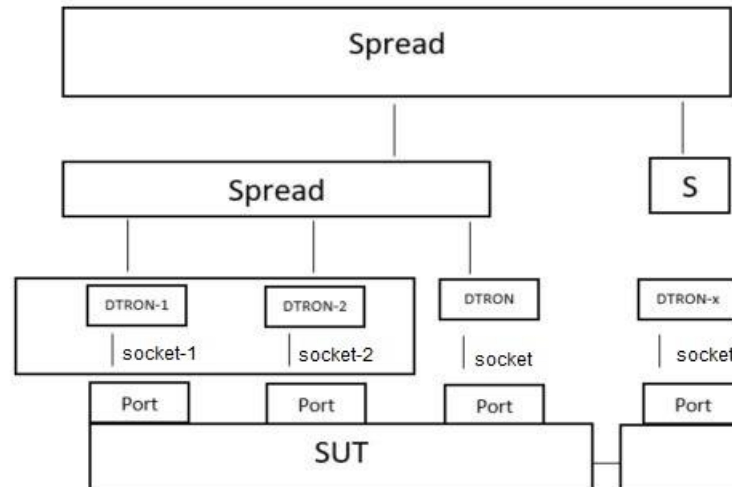

Figure 11: Distributed testing data-flow in DTron [22]

Figure 11 shows a conceptual view of the distributed runtime configuration of DTron. As we can see from the figure above, in the bottom, there is a set of distributed SUT components and each SUT component has a port for stimuli and observations [22]. Each port is connected to a DTron instance which can be an Adaptor, a Model or a combination of both. The DTron instance communicates over Spread which can be clustered. The flow of the tests is such that the local testers are integrated into the DTron instances which in turn is connected to the SUT ports via Adaptors and then subscribed to the Spread broker it is linked to.

DTron automatically generates the adaptor which gets linked to Spread multicast network. The low-level data (in byte form) which are transmitted in the Spread Network get serialized / de-serialized with the help of Google Protocol Buffers.

3 Case Study

This section has a short description of the satellite and its architecture which is the building block of this thesis. The satellite's architecture is also explained along with the component which has been chosen as the System Under Test.

3.1 Description of Satellite

The TUT nanosatellite is a student satellite comprising of various components. It is a project funded by the Tallinn University of Technology and various industries. The development and making of the satellite have been done since 2016, and most likely the satellite is proposed to be launched in the year 2018.

The main goals and objective of the nanosatellite are:

- Hands-on practical experience for students who can implement their knowledge and skills related to satellites and space.

- To build a satellite which can be operated from the Earth station and can be used for various productive reasons.

3.2 Satellite Architecture

The generic architecture of the TUT nanosatellite can be seen from the following Figure 12.

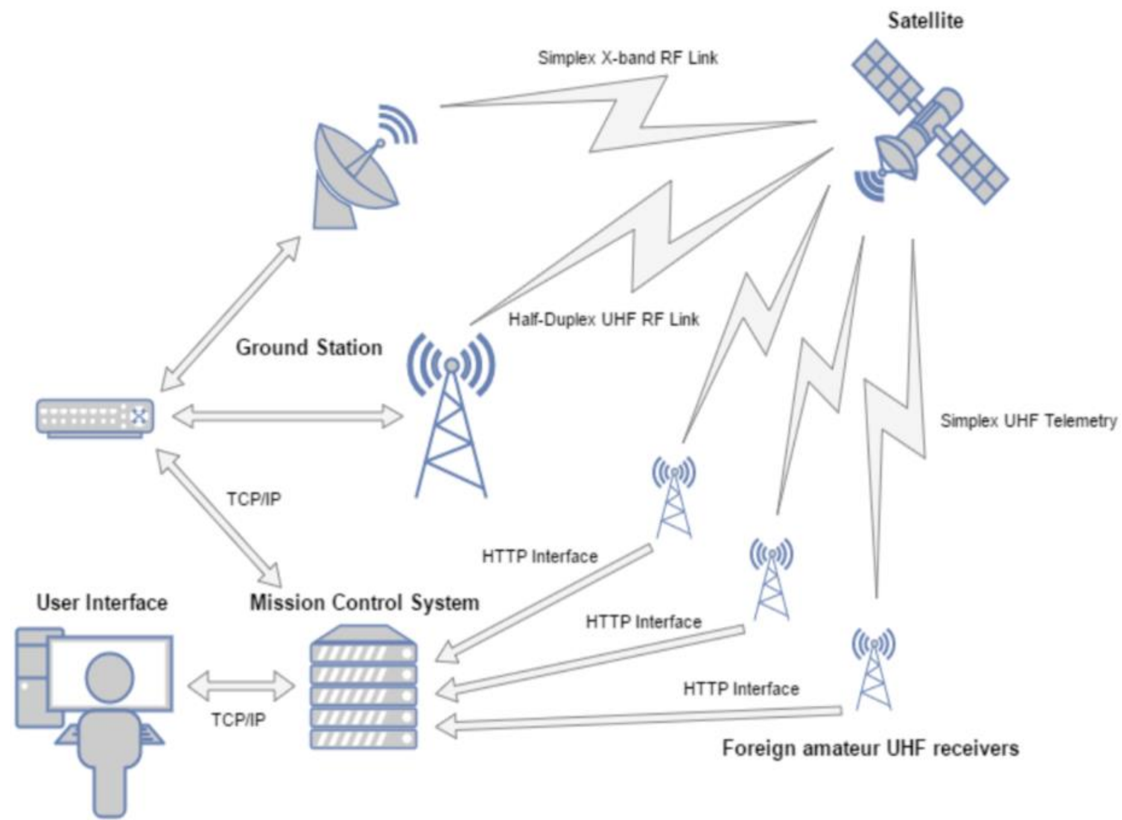

Figure 12: TUT Mektory Nanosatellite System [23]

The satellite system has both ground and space components that interact with each other to establish a well-defined communication and functional integrity for all phases of the mission.

The satellite system can be broadly divided into two main segments: *Ground Segment* and *Space Segment*.

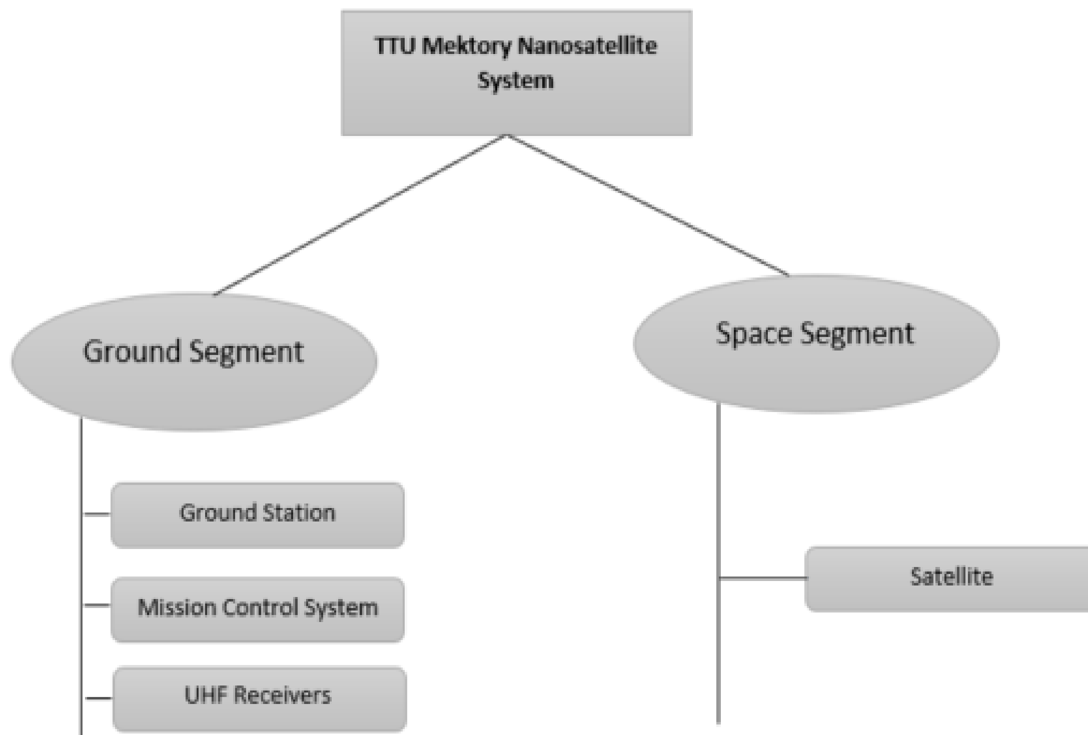

Figure 13: TTU Mektory Nanosatellite System Segments

The Ground Segment is further divided into *Ground station*, *Mission Control System* and the foreign amateur *UHF (Ultra High Frequency) receivers*. The *Space Segment* consists of the *Nanosatellite* itself.

3.3 System Under Test

In this thesis, the communication protocol between the ground station and the satellite is studied and tested. The following figure shows the communication protocol between the satellite and the ground station. A number of AX.25 protocol compatible frames is sent from the ground station to the satellite, and the satellite responds with a number of AX.25 frames within a time slot. The TUT nanosatellite supports AX.25 UI frames (Unnumbered Information).

Figure 14: TTU Mektory Nanosatellite Communication Protocol [23]

AX.25 is a data link layer protocol which is designed for use by the amateur radio operators (use of radio frequency spectrum for the purpose of exchanging messages via wireless or other means) [25].

4 Application of the Methodology and Tool Chain

The main implementation of the Satellite Communication Protocol and the modeling of the System Under Test is elaborated in this section. Tools like Dtron and Uppaal will be used to create the model and then validate it along with the generation of tests.

4.1 Class Diagram of the System Under Test

The System Under Test is first represented using a Class Diagram where the Adaptor has been introduced as a channel which will allow the test cases to run over the satellite software and the result obtained will be finally verified to be correct.

The main components of the class diagram are based on the nanosatellite TM/TC Protocol Description [23].

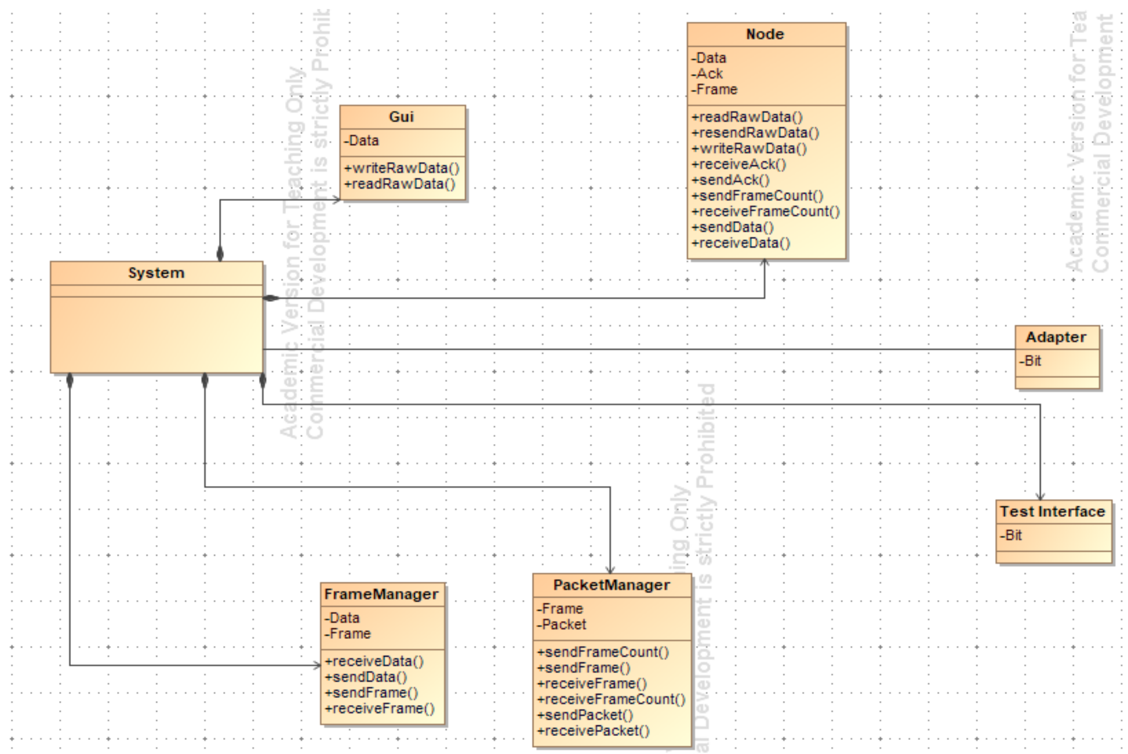

Figure 15: Class Diagram of Satellite Communication Protocol [24]

In Figure 15, the system consists of five components: *GUI, Node, Test Interface, Frame Manager* and *Packet Manager*. The *Adaptor* is used to connect to the System so that test cases can be run against the system. Each component of the Satellite is used to send and receive some Data.

The communication is initiated by *GUI* component. The *GUI* component can be considered as an environment component to SUT which generates messages to the satellite and processes acknowledgements from it. The messages are kept in the buffer until the positive acknowledgement has been received about its delivery [23]. The data is sent from the *GUI* to the *Node*. The communication is managed by the *Node*. The count of the data which is divided into L3 data frames is sent to the *Packet Manager*. Then, the frames are packed into the *Info* Object and *Operation Data* field and they are then sent to the *Frame Manager* [23]. This *Frame Manager* is responsible for packing the *Info* Object into an AX.25 frame. Since the *Packet Manager* has already received the count of the packets, so it collects the frames and assembles them in a packet called the *burst* which is finally sent to the satellite. When the satellite sends data, then it is received in an inverted process, and the data gets represented in the *GUI.* As the *Node* manages the communication, it then accumulates all the L3 data acknowledgment frames containing information about the frames that were not received by the satellite so that it can be used to resend it [23].

There are three types of communication situations that can occur [23]. These are explained below:

- Success – This is the ideal situation where the data that is sent to the satellite is received successfully.
- Semi-Success – There is a failure in proper communication due to some frames being lost or timeout. In this situation, after the reception, the L3 acknowledgment frame is sent which involves missing frame numbers. This leads to a failure compensation in communication.
- Timeout – A timeout can occur anytime due to some technical error thus leading to some frames being lost. Whenever there is a timeout, then the sender requests

for the frame numbers that were not received in the next communication window. It then resends the frames based on the L3 acknowledgment frame received.

4.2 Satellite Communication Architecture

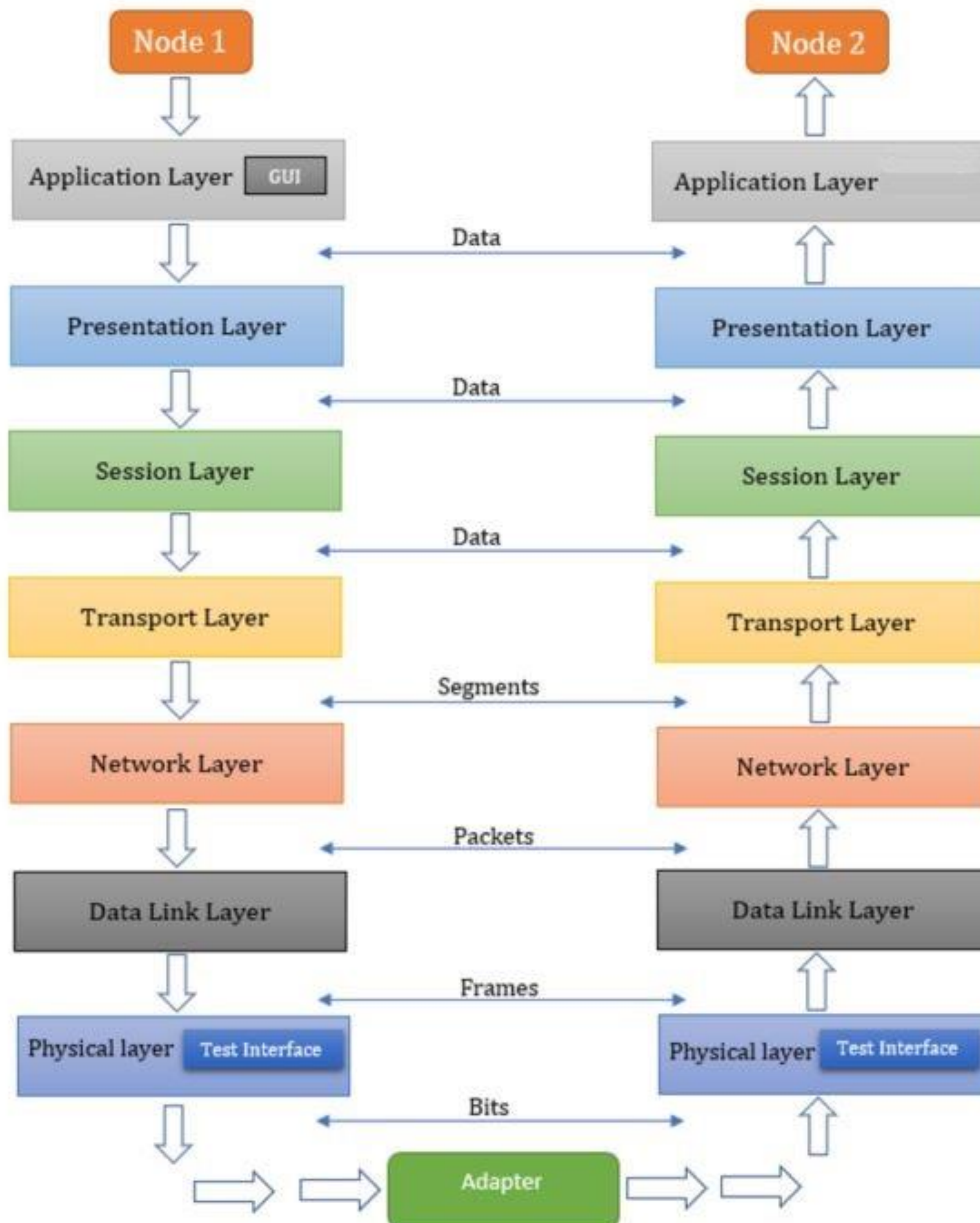

Figure 16: Satellite Communication Protocol Architecture

The satellite communication architecture can be considered like the diagram specified in Figure 16 where the two Nodes represent the Sender (Node 1) and the Receiver (Node 2).

The above diagram conforms to standard OSI model (Open Systems Interconnection Model which is used telecommunication).

Initially, the Sender (Node 1) initiates sending the data in the form of data from the GUI. This data travels from the GUI and follows the OSI layer protocol traveling from the Application layer to the Physical layer where the Adaptor helps to transmit data from the Sender to Receiver. Once the receiver receives the data in the physical layer, then the same workflow continues but in the reversed order until it reaches the satellite.

4.3 SUT Modelling using Uppaal

The modeling of the SUT is done with the help of Uppaal modeling tool. At first, a simple diagram is made to illustrate the SUT and the communication protocol. This diagram is shown in Figure 17. Here the test interface is denoted by dotted line. There are four test interface ports:

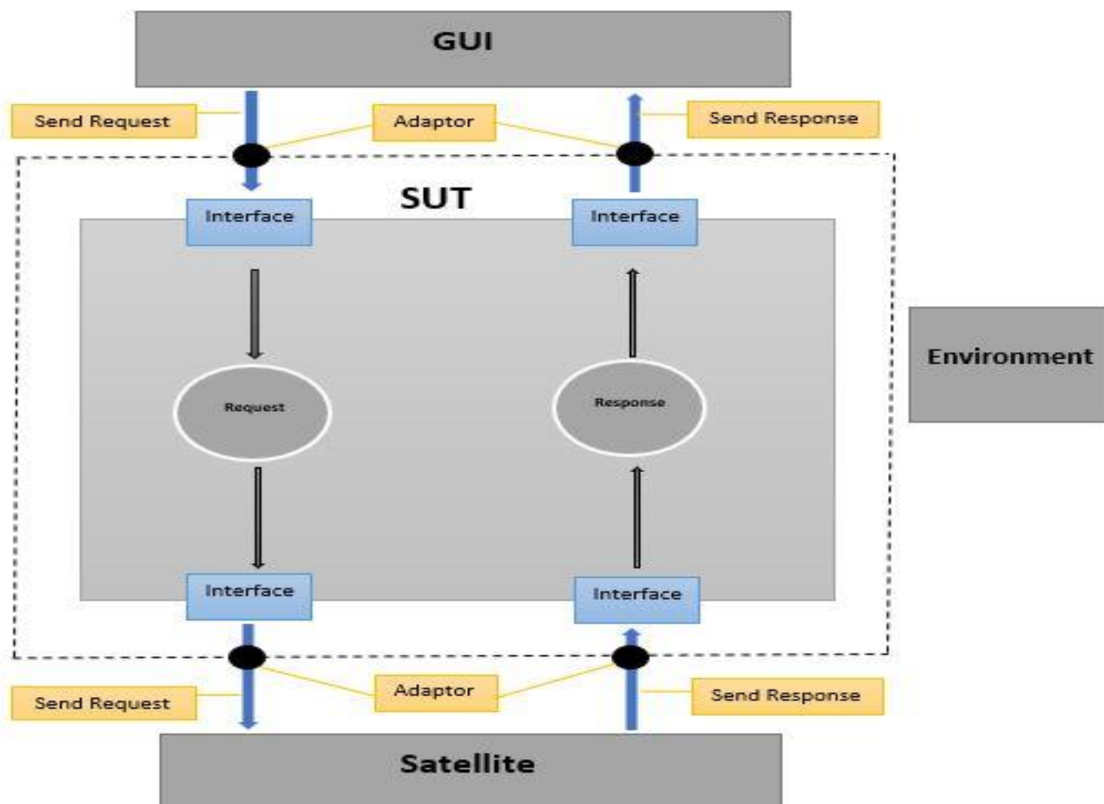

Figure 17: Satellite Communication Protocol

From Figure 17, we see that *GUI* initiates the communication and this data is sent from the *GUI* to the *Satellite* via the *SUT* which has interfaces that help to transmit data. Once the *satellite* receives the request data, it sends back the response data which is then received by the *GUI*.

With the above idea we model the interaction observable on test interfaces as shown in Figure 18:

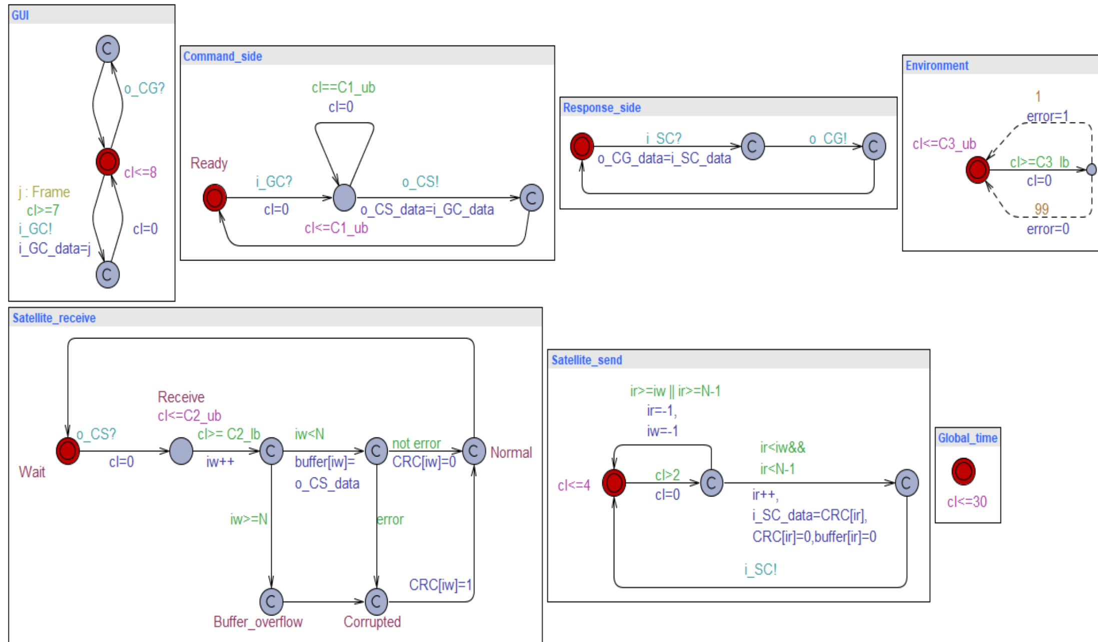

Figure 18: Nanosatellite Model using Uppaal Tool

In the above model, we have created following Uppaal TA templates:

- <u>GUI</u>: The GUI generates data frames that have type identifier "j" and that has assigned a random value in the range defined by type name 'Frame'. The data/request is first communicated to the control system process template 'Command_side' by the GUI, and once the satellite has received the request, the satellite responds and sends the response data back to the GUI via control system process template 'Response_side'. For sending the channel i_GC is used and for receiving channel o_CG is used.

- <u>Command_side</u>: The template Command_side forwards the data (the value of variable i_GC_data received via i_GC) to the process template 'Satellite_receive', via channel o_CS that has data variable o_CS_data attached to it.

- Response_side: The response side is where the response from the satellite is received and transmitted then to the GUI. The transmission of response data is from the SUT to the GUI (represented as *o_CG*) after the SUT has received the response from the satellite (channel i_SC).

- Satellite_receive: The template Satellite_receive stores the received data in the FIFO buffer (array '*buffer'* of size *N*) since the stream of incoming data via channel o_CS may not always be synchronised with the stream of responses (sent via channel i_SC). These streams are processed with different processes (in the model the template Satellite_receive and Satellite _send) that have their own pointers to the data item currently processed in the buffer. Explicit setting of the buffer size *N* is necessary to verify possible buffer overflow situations in case of transmission errors or asymmetric processing speeds between receiving commands and sending responses.

- Satellite_send: The template Satellite_send picks the oldest data item in the FIFO buffer, generates and sends response data back to SUT via channel i_SC.

- Environment: The template 'Environment' is to model the stream of transmission errors. That is to introduce real-world scenario considering the fact that the data may not be successfully transmitted or received due to various factors such as a timeout or technical issues anytime during transmission. The average error rate is modelled using stochastic extension of Uppaal TA, this is using probabilistic transitions denoted by dotted lines and numeric values of probabilities (current example specifies the error rate 1 percent of all transmitted frames).

- Global_time: To set some limit to the state space generated for verifying safety properties we have introduced the process template 'Global_time'. The clock invariant introduced in the template defines the upper time bound until which the behaviours of the model are observed. Without this time bound both the simulation and model checking would go infinitely.

4.3.1 Correctness of the Model

After the modeling is done using the Uppaal tool, we will verify the correctness of the model by checking for various properties [26].

- Connected: The model is said to be connected if there exists an executable path from one location to any other location and the query to check it is -> *'A[] not deadlock.'*

  From our model, we observe that Uppaal tool states that the '*Property is not satisfied*.' This is because we introduced *Global Time* which introduces a deadlock otherwise the execution will go on for infinity.

- Input Enabledness: Any test input during test execution should not cause blocking. This property holds true in the model.

- Strong Responsiveness: - No livelock (a loop that includes only ε -transitions) is reachable in the test model, i.e., the quiescent state is always reachable after ≤ m steps for some finite m [26]. To ensure this property is satisfied two steps need to be completed:

  - *Step* 1: Define auxiliary initial location $l0'$ and edges ($l0, li$), $i = 1, n$, where (n – number of locations in the SUT model).
  - *Step* 2: Model check the query

  $M^{SUT} \,||\, M^{Env}, l0' \models A\langle\rangle\ \exists(i{:}int\ [1,n])\, quiescent(....li)$,

Where M^{Env} denotes the model components of SUT environment. The predicate *quiescent* is implemented by introducing a Boolean array the elements of which are updated to *true* when input labelled edges are executed.

4.3.2 Test Purpose Specification

Test Purpose Specification states what is the goal of given test case and how to estimate the coverage achieved by the test run. It may refer to test coverage items or the scenario of choosing test stimuli [26]. In this thesis the test purpose is specified in English language, at first. Then it is formalized by mapping the informal description to the combination of test model structural constraints and the constraints expressed in model checking query. Both are usually needed for efficient test sequence generation and to prevent the state space explosion when running the generative model checking query.

The model can be updated by its refinement in a way that instead of all legal behaviours only those of interest will be preserved. Another option is to restrict the model of environment so that the test stimuli of interest are enabled. The changes are made in the *Environment* model. Once the changes are implemented, the formal model checking query is composed and executed. If the model is restrictive enough the query may have relatively simple form, e.g. E<> *target_state*, where *target_state* denotes the final state the searched test sequence has to reach.

There are *four* test purpose specifications implemented for the case study with two being *Successful* cases and the other two being the *Unsuccessful* cases where the messages are not delivered or received correctly.

The test purpose specifications are explained further as follows.

For test purpose specification we apply the approach where *Environment* template is refined with constraints so that test input sequences of interest are only generated. The *SUT* model itself *(Command_send* and *Command_received)* is left unchanged.

1) **Test Purpose 1:** Cover the situation where the data/frames are successfully received by the Satellite

This test purpose is where we consider the situation where the data/frames that are sent by the *GUI* are successfully received by the *Satellite*.

The model is updated in the following templates to address this test purpose:

- *GUI*: A new location is introduced to capture the situation where a single data item is sent by GUI and after that it makes transition to this terminal location.

- *Satellite_receive*: We distinguish between the normal and the corrupted data by introducing a new location called *Abnormal*. So, whenever the data received by the satellite is corrupted, we can easily distinguish it by seeing that the Cyclic Redundancy Check is set to 1 (*CRC[iw]=1*).

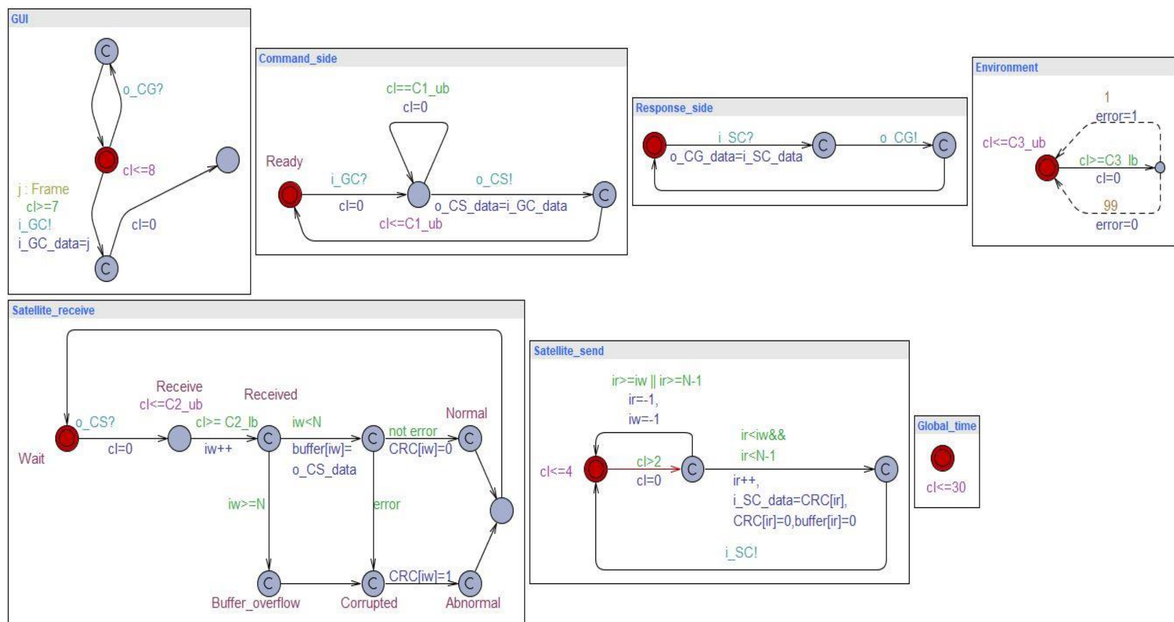

Figure 19: Test Purpose 1 Model

Query: E<> Satellite_receive.Normal && buffer[0]==i_GC_data

The above query implies that there exists a model run such that satellite receives the data sent from the GUI successfully (*Satellite_receive.Normal*) and that the message buffer in satellite involves this data (*buffer[0]==i_GC_data*). When we run the query in the *Verifier* of Uppaal, we see that the property gets satisfied.

```
E<> Satellite_receive.Normal && buffer[0]==i_GC_data
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 7,172KB / 26,452KB.
Property is satisfied.
```

Figure 20: Test Purpose 1 Verification

Witness Trace:

The witness trace generated by model checker gives a clear idea of which transitions were taken in which order for the query to be satisfied on the model. It helps to analyze the time stamps for the inputs and the values of the variables communicated between SUT and Environment model.

Figure 21: Test Purpose 1 Witness Trace (Step by Step)

2) **Test Purpose 2:** Test if the satellite sends the response and it gets received by the GUI successfully

Here we consider the situation where the data/frames that are sent by the *GUI* is successfully received by the *Satellite,* and the response from the *Satellite* is also received successfully by the *GUI*.

The model is updated in the following templates to address this test purpose:

GUI: There is a new *Buffer* (*GUI_buffer*) introduced which stores the information that the message or data that had been sent by the *GUI* has been successfully received by the *Satellite* the response to the same data from the *Satellite* has been received by the *GUI.* Therefore, there is a message counter which keeps count of the messages that have been sent by the *GUI*. Since it is assumed that at most five messages can be sent before getting the first acknowledged it is modulo five counter.

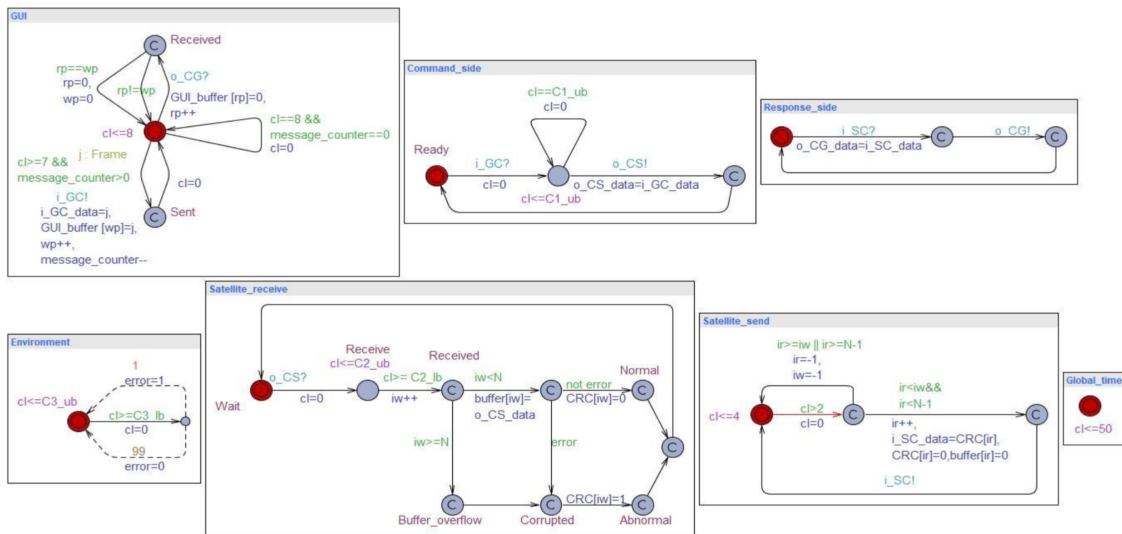

Figure 22: Test Purpose 2 Model

Query: E<> forall (j:history) (GUI_buffer[j]==0) && Global_time.cl==50

The above query implies that for all the messages sent, the responses have been received by the *GUI.* It is due to GUI model property that sent messages buffer element is reset to 0 when the acknowledgment about its successful transmission has been received. So, once the responses to the messages have been received, the *GUI_buffer* is set to zero and the maximum time interval for all the responses to come is 50 clock time intervals.

```
E<> forall (j:history) (GUI_buffer[j]==0) && Global_time.cl==50
Verification/kernel/elapsed time used: 6.344s / 0.14s / 6.515s.
Resident/virtual memory usage peaks: 31,020KB / 64,296KB.
Property is satisfied.
```

Figure 23: Test Purpose 2 Verification

Witness Trace:

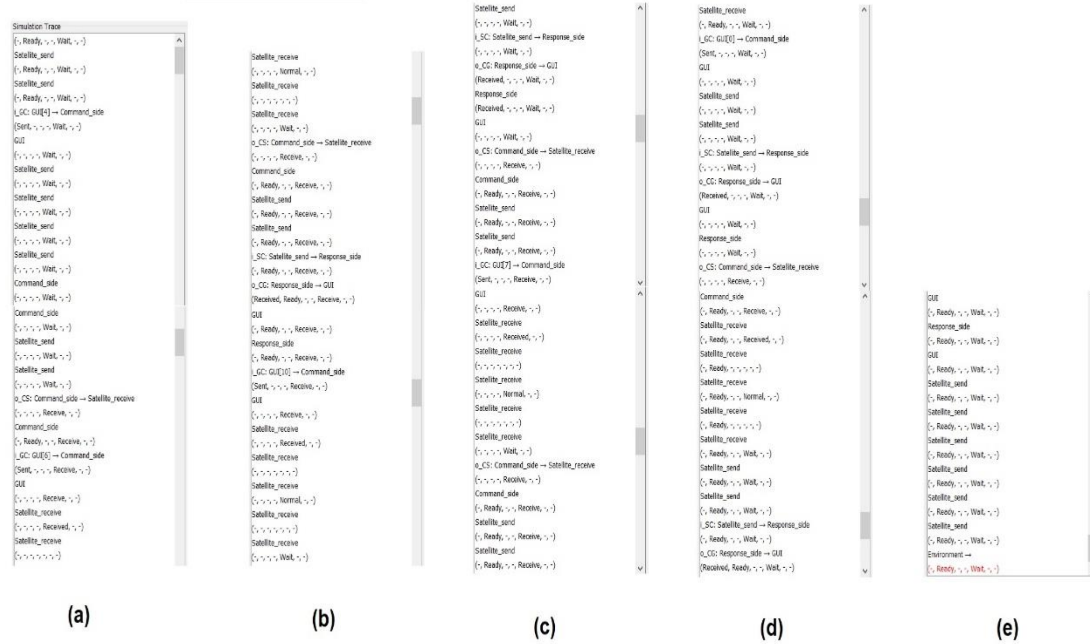

Figure 24: Test Purpose 2 Witness Trace (Step by Step)

In the above witness trace, we can see the entire transition for five messages which are being sent and then responses being received by the GUI during the global clock 50 time units.

3) **Test Purpose 3:** Data/Frames getting lost during transmission due to technical issues or timeout

This test purpose is where the data/frames being sent by the *GUI* is being lost during its transmission to the *Satellite.* This loss of data is due to technical issues or a timeout.

To address this test purpose the model is modified as follows:

- *GUI:* We introduce a Boolean variable '*nack'* which is the acknowledgment of incorrectly received frame/data sent from the satellite. There are also two new locations introduced called '*Ack'* (The location which is reached when there is no corrupted data acknowledgment) and '*Nack'* (The location which is reached when there is corrupted data acknowledgment). Additionally,

variable ‘T’ is for the Trap which is used to represent that there is a corruption of data. So, when we get a negative acknowledgment from the satellite meaning that there is either a timeout, or some technical issues occurred while the data was being transmitted from *GUI* to *Satellite*, the variable ‘T’ is set to 1.

- *Environment:* We increase the possibility of the occurrence of an error, and the ratio is set to 1:1 to address the test purpose.

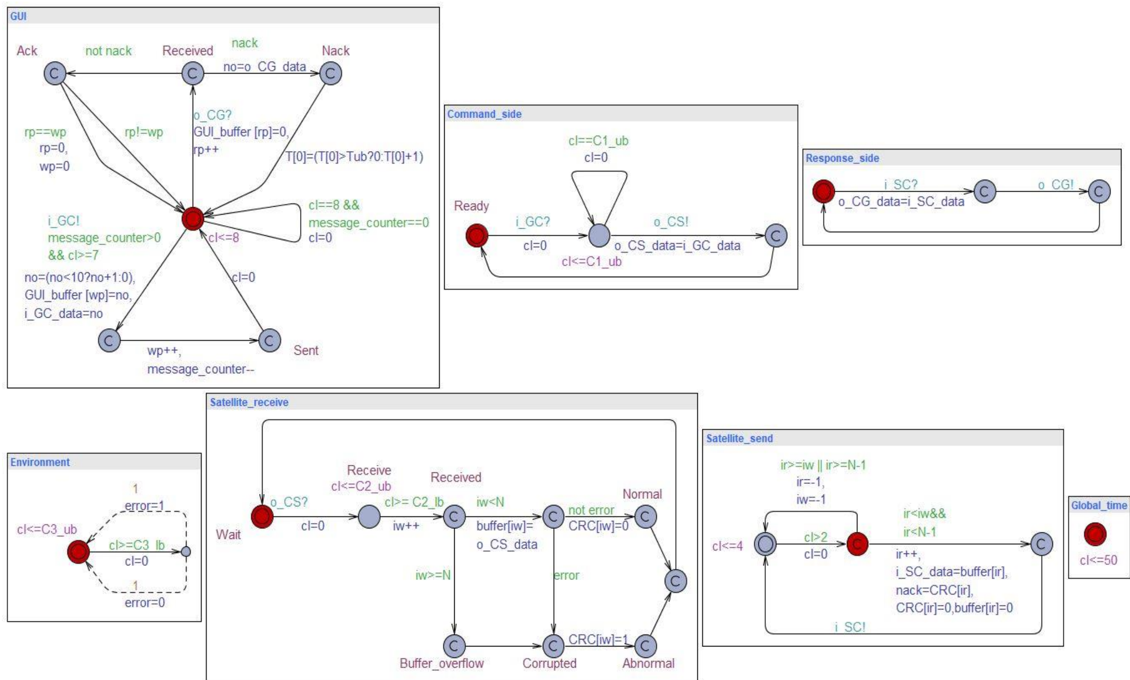

Figure 25: Test Purpose 3 Model

Query: E<> T[0]==1

The above query specifies that there is one corrupted response/ acknowledgment for 10 messages (arbitrary value) sent by the *GUI* and the frames/data being sent is lost due to the error (in the above case it is a timeout).

```
E<> T[0]==1
Verification/kernel/elapsed time used: 0.125s / 0.015s / 0.129s.
Resident/virtual memory usage peaks: 7,744KB / 27,332KB.
Property is satisfied.
```

Figure 26: Test Purpose 3 Verification

Witness Trace:

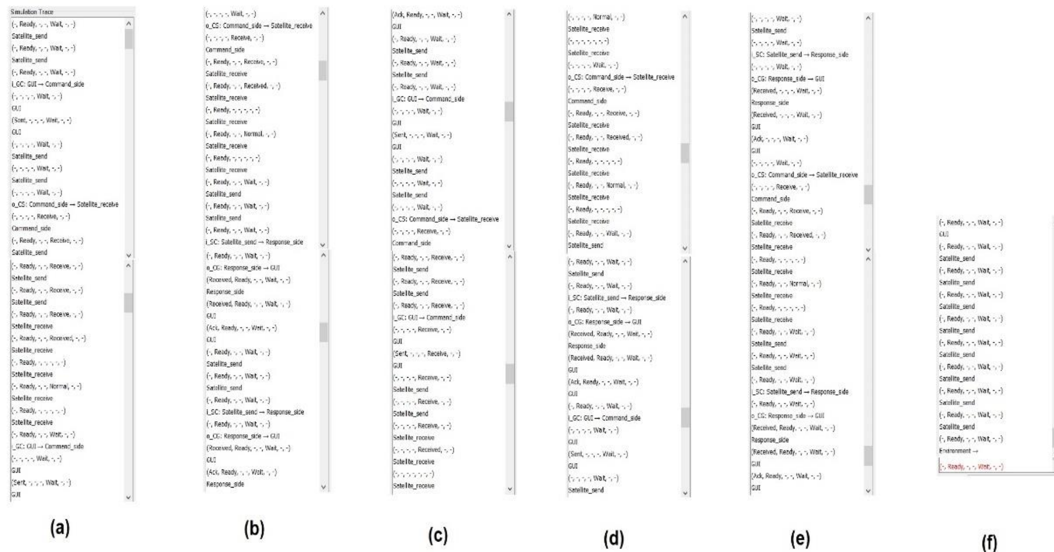

Figure 27: Test Purpose 3 Witness Trace (Step by Step)

In the above trace, we see the transmission of ten messages which are being sent and then responses being received by the GUI for a global time clock of 50 time units. After the timeout where the Global Time clock exceeds 50, then an error is introduced from the *Environment* which leads to the frames getting lost and a negative acknowledgment from the *Satellite*.

4) **Test Purpose 4:** <u>Satellite sending a late response to a particular request making it hard to know what the actual request was</u>

This test purpose identifies if the response from the Satellite for the first request is delayed and the second or other responses are sent to the *GUI* before it. This makes the user on the *GUI* end difficult to understand which response relates to which request made by the *GUI*.

The model is updated in the following templates to address this test purpose:

- *Satellite_send*: We introduce a time delay in one of the Acknowledgement responses from the *Satellite*. This is done by making the read pointer ('*ir*') read the second value and skip the first ('*ir1=ir+2*').

- *GUI*: Clock is set to zero ('*cl=0*'). When a response is received by the *GUI*, the clock is reset also meaning that the incoming responses do not take time.

- *Global_time:* The global time is increased from 50 to 100.

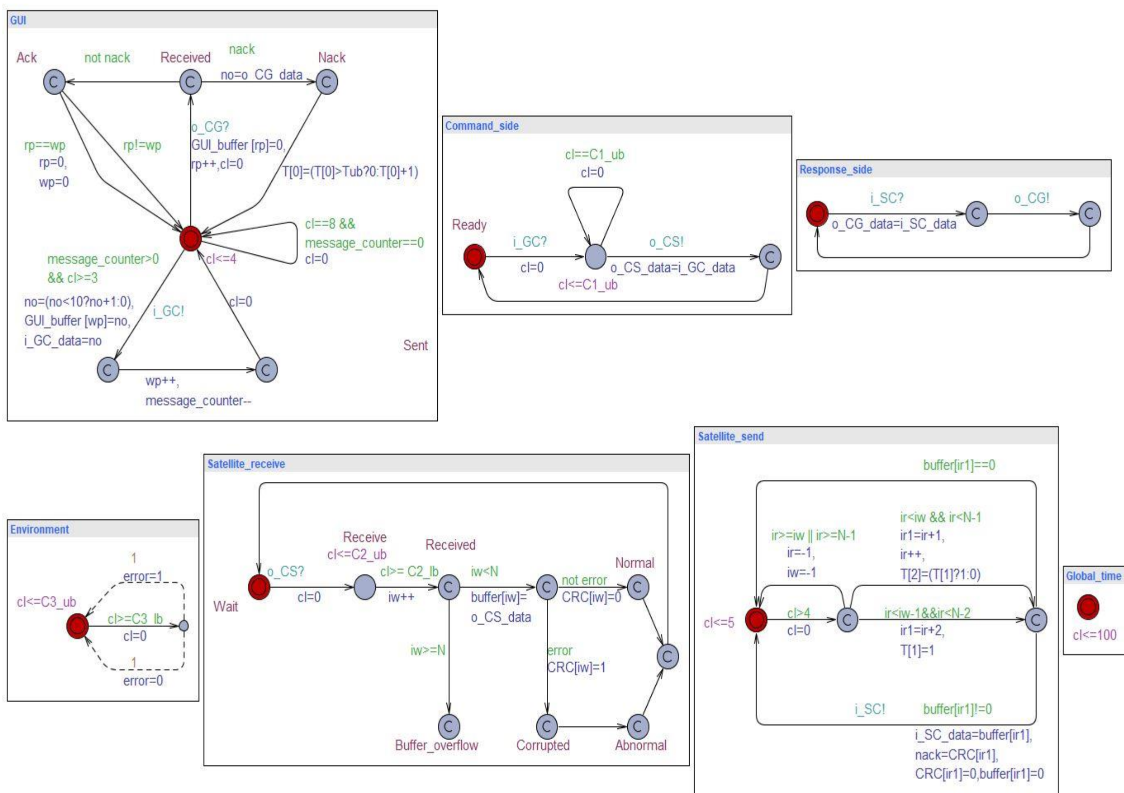

Figure 28: Test Purpose 4 Model

Query: E<> T[0]==1, E<> T[2]

There are two queries written for this test purpose. The first one specifies that trap will be covered indicating that the issue exists and is reachable in the model, and the second response was sent before the first one. The second query verifies that it is inevitable that there was a late response to a particular request.

E<> T[2]
Verification/kernel/elapsed time used: 0s / 0s / 0.006s.
Resident/virtual memory usage peaks: 7,184KB / 26,488KB.
Property is satisfied.

Figure 29: Test Purpose 4 Verification

Witness Trace:

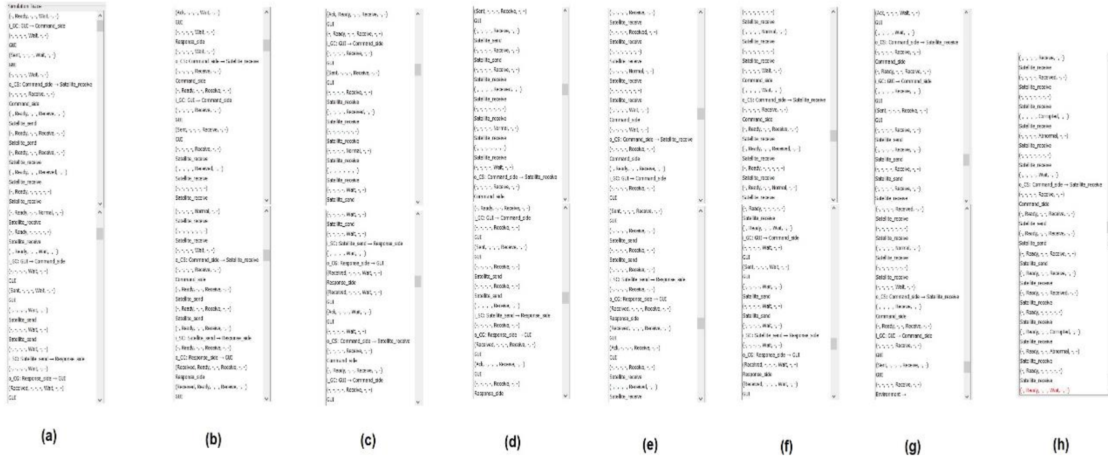

Figure 30: Test Purpose 4 Witness Trace (Step by Step)

In the above witness trace, we see that a situation is reachable where the *Satellite* sends a delayed response to a request and to capture it; we increased the *Global Time* from 50 to 100. Thus, we can see what happens when a delayed response is sent by the satellite.

In addition to proving the existence of witness traces to satisfy the test purposes Uppaal model checker allows to optimize them as well. There are two options for trace generation either shortest or fastest trace. First means that the number of transitions in the trace is minimal and the second that the trace is executed in shortest time.

5 Analysis and Validation of Results

To evaluate the feasibility of the approach an analysis has been done on time taken to model the SUT and on the time taken for specification and execution of every test purpose. It is estimated that the modeling is done under expert supervision which otherwise would have taken much longer time.

5.1 Time Taken to Model the SUT

The modeling of the communication protocol took plenty of time which involved discussions with developers, and others who are involved in the project currently to know better about the Satellite and its functionalities.

Table 1: Time Taken for Modelling SUT

Meetings	Time Taken (in hours)
Project Introduction (Code)	3
Discussion and understanding about the communication protocol	4
Modeling using Uppaal	4

5.2 Time Taken to Update Model for Test Purpose Specification

The following table shows how much time it took to update the model to get the test purpose addressed and verified.

Table 2: Time Taken for Test Purpose Specification and Verification

Serial No.	Test Purpose	Time Taken (in hours)
1	Cover the situation where the data/frames are successfully received by the Satellite	0.75
2	Test if the satellite sends the response and it gets received by the GUI successfully	1
3	Data/Frames getting lost during transmission due to technical issues or timeout	1.5
4	Satellite sending a late response to a particular request making it hard to know what the actual request was	1

5.3 Timed Trace Analysis

The model checking witness trace for every test purpose specification is analyzed below:

Timed Trace for Test Purpose Specification 1: Cover the situation where the data/frames are successfully received by the Satellite

Table 3: Timed Trace for Test Purpose Specification 1

Source -> Destination	Message Type (I/O)	Data (data values)	Relative time stamps (in model time units)
GUI -> Command_side	i_GC	0	[7,8]
Command_side -> Satellite_receive	o_CS	0	[7,8]
Satellite_send -> Response_side	i_SC	0	[11,15]
Total test execution time			[25, 31]

Timed Trace for Test Purpose Specification 2: Test if the satellite sends the response and it gets received by the GUI successfully

Table 4: Timed Trace for Test Purpose Specification 2

Source -> Destination	Message Type (I/O)	Data (data values)	Relative time stamps (in model time units)
GUI -> Command_side	i_GC	0	[7,8]
Command_side -> Satellite_receive	o_CS	0	[7,12]

Satellite_send -> Response_side	i_SC	0	[9,12]
Response_side -> GUI	o_CG	0	[9,12]
GUI -> Command_side	i_GC	6	[14,16]
Command_side -> Satellite_receive	o_CS	6	[14,16]
Satellite_send -> Response_side	i_SC	0	[16,16]
Response_side -> GUI	o_CG	0	[16,16]
GUI -> Command_side	i_GC	7	[21,22]
Command_side -> Satellite_receive	o_CS	7	[22,26]
GUI -> Command_side	i_GC	5	[28,29]
Satellite_send -> Response_side	i_SC	0	[28,30]
Response_side -> GUI	o_CG	0	[28,30]
Command_side -> Satellite_receive	o_CS	5	[32,37]
GUI -> Command_side	i_GC	0	[35,37]
Command_side -> Satellite_receive	o_CS	0	[35,37]
Satellite_send -> Response_side	i_SC	0	[35,37]

Response_side -> GUI	o_CG	0	[35,37]
Satellite_send -> Response_side	i_SC	0	[43,44]
Response_side -> GUI	o_CG	0	[43,44]
Total test execution time			[442,518]

Timed Trace for Test Purpose Specification 3: Data/Frames getting lost during transmission due to technical issues or timeout

Table 5: Timed Trace for Test Purpose Specification 3

Source -> Destination	Message Type (I/O)	Data (data values)	Relative time stamps (in model time units)
GUI -> Command_side	i_GC	0	[7,8]
Command_side -> Satellite_receive	o_CS	0	[7,12]
Satellite_send -> Response_side	i_SC	1	[12,16]
Response_side -> GUI	o_CG	1	[12,16]
GUI -> Command_side	i_GC	1	[14,16]
Command_side -> Satellite_receive	o_CS	1	[14,20]
GUI -> Command_side	i_GC	2	[21,23]

Satellite_send -> Response_side	i_SC	2	[22,27]
Response_side -> GUI	o_CG	2	[22,27]
Command_side -> Satellite_receive	o_CS	2	[25,31]
GUI -> Command_side	i_GC	3	[28,31]
Command_side -> Satellite_receive	o_CS	3	[28,31]
Satellite_send -> Response_side	i_SC	3	[28,31]
Response_side -> GUI	o_CG	3	[28,31]
GUI -> Command_side	i_GC	4	[35,38]
Satellite_send -> Response_side	i_SC	4	[35,38]
Response_side -> GUI	o_CG	4	[35,38]
Command_side -> Satellite_receive	o_CS	4	[35,42]
Satellite_send -> Response_side	i_SC	5	[43,49]
Response_side -> GUI	o_CG	5	[43,49]
Total test execution time			[466,574]

<u>Timed Trace for Test Purpose Specification 4:</u> Satellite sending a late response to a particular request making it hard to know what the actual request was

Table 6: Timed Trace for Test Purpose Specification 4

Source -> Destination	Message Type (I/O)	Data (data values)	Relative time stamps (in model time units)
GUI -> Command_side	i_GC	0	[3,4]
Command_side -> Satellite_receive	o_CS	0	[3,5]
GUI -> Command_side	i_GC	1	[6,8]
Command_side -> Satellite_receive	o_CS	1	[10,10]
Satellite_send -> Response_side	i_SC	0	[10,10]
Response_side -> GUI	o_CG	1	[10,10]
GUI -> Command_side	i_GC	2	[13,13]
Command_side -> Satellite_receive	o_CS	2	[13,15]
Satellite_send -> Response_side	i_SC	1	[15,15]
Response_side -> GUI	o_CG	2	[15,15]
GUI -> Command_side	i_GC	3	[18,19]
Command_side -> Satellite_receive	o_CS	3	[18,20]

Satellite_send -> Response_side	i_SC	2	[20,20]
Response_side -> GUI	o_CG	4	[20,20]
GUI -> Command_side	i_GC	4	[23,24]
Command_side -> Satellite_receive	o_CS	4	[23,25]
Satellite_send -> Response_side	i_SC	4	[25,25]
Response_side -> GUI	o_CG	3	[25,25]
GUI -> Command_side	i_GC	5	[28,29]
Command_side -> Satellite_receive	o_CS	5	[29,33]
GUI -> Command_side	i_GC	6	[31,33]
Satellite_send -> Response_side	i_SC	3	[33,35]
Response_side -> GUI	o_CG	5	[33,35]
Command_side -> Satellite_receive	o_CS	6	[33,37]
GUI -> Command_side	i_GC	7	[36,39]
Command_side -> Satellite_receive	o_CS	7	[36,40]
Satellite_send -> Response_side	i_SC	5	[37,40]

Response_side -> GUI	o_CG	6	[37,40]
GUI -> Command_side	i_GC	8	[40,43]
Satellite_send -> Response_side	i_SC	6	[41,45]
Response_side -> GUI	o_CG	7	[41,45]
Command_side -> Satellite_receive	o_CS	8	[41,47]
GUI -> Command_side	i_GC	9	[44,49]
Command_side -> Satellite_receive	o_CS	9	[50,50]
Satellite_send -> Response_side	i_SC	7	[50,50]
Response_side -> GUI	o_CG	8	[50,50]
Total test execution time			[960,1023]

5.4 Test Generation Time and Memory Usage Analysis

This section shows the time taken and the memory used to verify query written for each Test Purpose specified.

Table 7: Time and Memory Usage Analysis

Serial No.	Test Purpose	Time Used (verification/kernel/elapsed time)	Memory usage (resident/virtual memory)
1	Cover the situation where the data/frames are successfully received by the Satellite	0s / 0s / 0.001s	7,172KB / 26,452KB
2	Test if the satellite sends the response and it gets received by the GUI successfully	6.344s / 0.14s / 6.515s	31,020KB / 64,296KB
3	Data/Frames getting lost during transmission due to technical issues or timeout	0.125s / 0.015s / 0.129s	7,744KB / 27,332KB
4	Satellite sending a late response to a particular request making it hard to know what the actual request was	0s / 0s / 0.006s	7,184KB / 26,488KB

5.5 Summary of Analysis and Validation of Results

From the above analysis, we see how long it takes to model the SUT and to verify the test purpose specifications. We also understand how we can update the model to address the test purposes specified. Alongside, the witness trace shows clearly which transition takes place at what time and how long it takes for a message to be delivered to the *Satellite.*

As an important conclusion we can claim that MBT can be compared with short test script writing time wise where SUT modelling and formalization of test purpose take the most of the test development time. This in turn confirms that MBT is more effective in regression testing where the test cases can be easily introduced incrementally by comprehensible model modifications. Generation of test sequences is fully automatic, and its time is negligible compared to requirements capture and formalization.

We also see the time and memory consumption for each test purpose. These observations and data can be further studied and analyzed.

6 Conclusion and Future Work

The main purpose of this thesis was to develop systematically the test cases for the TUT Mektory Nanosatellite and prove them to be correct. The model-based approach involves modeling the System Under Test in its early development phases by capturing some of the requirements whenever they are available for testing.

The author diligently followed the Provably Correct Test Development Workflow and started step by step with first modelling the SUT which most certainly helped to envision the satellite communication protocol precisely, followed by specifying the test purpose which had four different scenarios of satellite communication, then generating the test based on the test purpose and finally executing it against the System Under Test. The results obtained from the tests gives a clear idea about the time taken to model the SUT, the time and memory used to execute the test purpose specifications, and the timed trace giving an overview about the timing constraints and the timelines.

In future, the author believes that the MBT process implemented based on the Uppaal tool family can be used successfully for further mission critical systems where time constraints and high degree of parallelism with extensive set of interaction between parallel components are the subjects of testing.

7 Acknowledgment

Firstly, I would like to express my deepest gratitude to my supervisor, Professor Jüri Vain for suggesting me an interesting topic and for continually encouraging and motivating me to write my thesis. Without his guidance and persistent help, this thesis would not have been possible.

Secondly, I would like to thank Evelin Halling for her time and for giving me an overview of the TUT Mektory nanosatellite and its functionality.

I would also like to thank Professors from Tallinn University of Technology and University of Tartu for their teachings and the opportunity to pursue Master's in Software Engineering.

Last but not the least, I would like to thank my parents and my brother for their unfailing support throughout my years of study.

References

[1] Mektory Nanosatellite Programme. https://www.ttu.ee/projects/mektory-eng/satellite-programme-3/ (accessed 2017-11-08).

[2] Utting, M., Legeard, B. (2006). Practical Model-Based Testing: A Tools Approach.

[3] Behrmann, G., David, A., and Larsen, K. G. A tutorial on Uppaal. In Formal Methods for the Design of Real-Time Systems (Bernardo, M. and Corradini, F., eds). Lecture Notes in Computer Science, Vol. 3185. Springer, Berlin, 2004, 200–236.

[4] DTRON. [WWW] https://cs.ttu.ee/dtron/ (accessed 2017-11-09).

[5] Bringmann, E., & Krämer, A. (2008). Model-Based Testing of Automotive Systems. In 2008 1st International Conference on Software Testing, Verification, and Validation (pp. 485–493). https://doi.org/10.1109/ICST.2008.45.

[6] Wagner, F., Dalton, S. W., & Bergmann, A. (2017). Benefits and drawbacks of target specific model-based testing. In 2017 International Conference on Research and Education in Mechatronics (REM) (pp. 1–5). https://doi.org/10.1109/REM.2017.8075249.

[7] Herpel, H. J., Kerep, M., Li, J., Xie, J., Johansen, B., Kvinnesland, K., … Barrios, P. (2016). Model based testing of satellite on-board software - An industrial use case. In IEEE Aerospace Conference Proceedings (Vol. 2016–June, pp. 1–9). https://doi.org/10.1109/AERO.2016.7500845.

[8] Uppaal Tron. [WWW] http://people.cs.aau.dk/~marius/tron/ (accessed 2017-11-09).

[9] Uppaal. [WWW] www.uppaal.org (accessed 2017-11-10).

[10] Altaf, I., Dar, J. A., Rashid, F. u., & Rafiq, M. (2015). Survey on selenium tool in software testing. In 2015 International Conference on Green Computing and Internet of Things (ICGCIoT) (pp. 1378–1383). https://doi.org/10.1109/ICGCIoT.2015.7380682.

[11] Vain J., Halling, E., Kanter, G., Anier, A., Pal, D. (2016) Model-Based Testing of Real-Time Distributed Systems. DB&IS: 272-286.

[12] Vain, J., Anier, A., & Halling, E. (2014). Provably correct test development for timed systems. Frontiers in Artificial Intelligence and Applications, Databases and Information Systems VIII-Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014, 289–302. https://doi.org/10.3233/978-1-61499-458-9-289.

[13] Enoiu, E. P., Sundmark, D., & Pettersson, P. (2013). Model-Based Test Suite Generation for Function Block Diagrams Using the UPPAAL Model Checker. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (pp. 158–167). http://doi.org/10.1109/ICSTW.2013.27.

[14] Gojare, S., Joshi, R., & Gaigaware, D. (2015). Analysis and Design of Selenium WebDriver Automation Testing Framework. Procedia Computer Science, 50(Supplement C), 341–346. https://doi.org/https://doi.org/10.1016/j.procs.2015.04.038.

[15] TestNG Documentation. [WWW]. http://www.testng.org. (accessed 2017-11-10).

[16] Vain, J., Kanter, G., Srinivasan, S. (2017). Model based testing of distributed time critical systems. Proceedings of 6th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), Noida, India, September 20-22, 2017. New Dehli, India: IEEE, 97−103.

[17] Hessel, A., and Others, Testing Real-Time Systems Using UPPAAL, Springer-Verlag, 2008, pp. 77–117.

[18] Vain, J., Truscan, D., Iqbal, J., Tsiopoulos, L. On the Benefits of Using Aspect-Orientation in UPPAAL Timed Automata.

[19] Uppaal Small Tutorial [WWW]. http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf. (accessed 2018-05-01).

[20] Vain, J., and Halling, E. Constraint-based test scenario description language.

[21] Vain, J., Kääramees, M., and Markvardt, M. Online Testing of Nondeterministic Systems with the Reactive Planning Tester.

[22] Anier, A., Vain, J., and Tsiopoulos, L. DTRON: A Tool for Distributed Model-Based Testing of Time Critical Applications.

[23] Rodionov, D., Halling, E. TMSS-SYS-TN-02-v0.5-Satellite-TCTM-Protocol-Description_1. (Not publicly accessible)

[24] Rodionov, D. Implementing TTU Nanosatellite Communication Protocol Using Taste Toolset.

[25] Amateur Radio. [WWW]. https://en.wikipedia.org/wiki/Amateur_radio, _https://en.wikipedia.org/wiki/AX.25. (accessed 2018-07-01).

[26] Vain,J. (2016). Distributed model based testing, IEEE CSS Winter School on Cyber-Physical, Systems 2016, Madurai, India.

[27] Behrmann,G., David, A., and G. Larsen, K. (2006). A Tutorial on Uppaal 4.0.

[28] Vain J., Halling, E., Kanter, G., Anier, A., Pal, D. (2016) Automatic Distribution of Local Testers for Testing Distributed Systems. DB&IS (Selected Papers): 297-310.

[29] P. Ernits, J., Vain J., Halling, E., Kanter, G. (2015) Model-based integration testing of ROS packages: A mobile robot case study. ECMR: 1-7 [WWW] (http://ieeexplore.ieee.org/document/7324210/).

[30] Anier, A. (2016) Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems.