Grading with TorXakis

by

Movahed Abdolahi

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
August 2023

Abstract:

Software development models and processes have become incredibly complex and sensitive since many governments and companies are implementing software to perform critical matters including such highly dangerous activities as surgeries. Evidently, with increasing sensitivity delivering robust and bug free software is getting more important than ever.

This thesis discusses the use of model-based testing to grade and validate various functionalities of a network application which in this case is a Bulletin Board System (BBS). In model-based testing test cases are automatically generated algorithmically to verify the correctness of the implementation of a system according to the model that describes the expected behavior of that system. We are intending to use TorXakis (a model-based testing tool) to grade 30 submissions to a final project in a graduate course at Bishop's University. We create a model according to the assignment handout and then we apply this model to the student submissions by using a MBT tool. The submissions will also be validated manually using telnet protocol [9] manually.. Comparing model-based testing results with human tested results we can see a high degree of similarity while automated grading is much more resource efficient.

Acknowledgments:

I am thankful to the Computer Science department of Bishop's University for granting me the opportunity to develop and pursue a Master's degree.

I wish to express my gratitude to Dr. Stefan D. Bruda, my supervisor, for his invaluable guidance and unwavering professional support during the entirety of this thesis. Without his assistance, this achievement would not have been possible.

Last but not least, I want to extend my heartfelt appreciation to my wife for her love and all the support that she provided, especially during the final phase of this thesis. Her constant encouragement and motivation were integral to my success in this endeavor.

Contents:

Chapter 1:

Introduction

Validation is one of the most important parts of software development. As clients' needs are getting more complex, the need for testing and being able to validate the components of a software is beginning to become even more obvious than before. It ensures that the software is working well and meets the requirements of the end user. Nowadays there are many methods to test a software product, even during the development stage.

Systematic software testing is one of the most important and widely used techniques to check the validity of a software. Testing however is often a manual and laborious process without effective automation, which makes it error-prone, time consuming, and very costly.

Two most important reasons that makes software testing so important is that it lowers the cost (in both time and money) spent on the development stages. In addition, software products released without proper testing can sometimes cause serious issues such as data leakage and even irrecoverable damage. On the other hand, by testing software thoroughly before releasing not only we can implement software in a shorter time and with less cost, but we can also ensure that the product has the expected quality and performance. In the end this all leads to a higher customer satisfaction rate which is the ultimate goal of the software life cycle.

Software testing used to be overlooked by companies and businesses but nowadays the importance of testing is no longer a mystery to anyone. Businesses are allocating a huge portion of their development budget to make sure the software is tested and performed as intended. Additionally, many companies are using methods such as the Software Production Line (SPL in short) [10], where it is even more important to make sure the best testing strategy is being used. This is getting substantial attention, for indeed it is a lot harder to test a Software production line. While software development and construction techniques enable the creation of more intricate systems, there is a real concern that testing methods may not be able to match the pace of software construction. This could potentially impede the advancement of future iterations of software systems.

One of the most effective software testing methods is model-based testing (MBT in short). MBT is an approach in which a model of the system under test is created and used to generate test cases and verify the output. The model represents the desired behavior and structure of the system (specification), and can be used to automate the testing process. It is often more precise than other methods and mainly used for testing more complex systems where manual testing is almost impossible and definitely not efficient.

From an industrial perspective, MBT is a promising technique to improve the quality and effectiveness of testing and to reduce its cost. From an academic perspective MBT is a compound of formal methods and verification techniques.

In this thesis we study the application of MBT to the more mundane task of grading assignments submitted by students. We first construct a model based on the specification given to students, and then we apply the model against all 30 submissions to grade the submissions based on their functional requirements. In this particular instance, model based testing (or testing in general) is a realistic marking technique, as the main task of the project was to create a program that observes a provided specification. Additionally MBT is capable of offering not just a binary yes/no responses, but also counterexamples, which can then be used to provide feedback on the respective submission. The specification asked for the implementation of a bulletin board system (BBS for short), including specific performance and concurrency requirements as well as facilities for data synchronization. We anticipate being capable of formulating a comprehensive model for all the functional prerequisites. Nevertheless, due to the inherent characteristics of MBT, examining performance or non-functional prerequisites through this approach remains unfeasible. As an additional facet of this thesis, we have explored the necessity of manually assessing non-functional requirements (concurrency requirements, robustness, system load, etc.) versus the possibility of deducing the accuracy of these non-functional requirements from the result of testing functional prerequisites. Interestingly, it seems that such an inference is possible. It should be noted however that one particular non-functional requirement namely, concurrency requirements can only be tested to the extent that they affect the functional behaviour.

In all, we wonder whether it is worth replacing manual grading with MBT. On one hand, MBT offers significant advantages over conventional

methods. It is notably faster and much more reliable as it does not require constant supervision during the testing phase. Unlike conventional methods which are prone to oversight, MBT can effectively prevent human errors during testing thus making the process fairer. On the other hand, we expect MBT to be a very tough grader.

We organize our thesis as follows. We first introduce MBT in general followed by an overview of the theory of testing and how process algebra made it possible. Next, we outline the specification given to the students for the project (a simplified BBS). Afterward we present the MBT tool that we used (called TorXakis) as well as the model developed from the given specification. Finally we discuss the process and outcome of applying our model to 30 BBS implementations by Bishop's University students as their project. We are therefore using MBT to validate their functionality as much as possible and provide a final grade for each submission, and then compare the grades thus achieved to conventional, human supervised methods.

1.1 Previous work

Over time, model-based testing (MBT) has been useful for specifying and validating diverse systems and protocols. Work in this area adopts distinct methodologies with regards to their application of MBT within their particular domain of focus. For instance, some MBT publications might emphasize specific aspects of MBT in the context of network protocols, while others may concentrate on applying MBT techniques to software systems. The variations in these approaches reflect the adaptable nature of MBT as it addresses the distinctive demands of each domain.

In other words, MBT can be applied (and has been applied) widely for the verification of computing systems. Recent examples include Tretmans and Van de Laar who applied MBT to validate the Dropbox protocol [2], Li, Pierce and Zdancewic who applied MBT on network applications [11], and Garous, et. al. who used MBT to validate Web applications [12].

To the best of our knowledge however MBT has not been used to assess academic projects.

Chapter 2:

Theory of model based testing

We now briefly review some notions related to the thesis. As mentioned earlier, model based testing is one of the most reliable methods for software testing. In this chapter we are exploring the theory of model-based testing and the underlying formal methods including process algebra and labeled transition systems.

2.1 Model-based testing (MBT)

Model-based testing is a form of black-box testing where a system under test (SUT for short) is tested against an abstract model of its required behavior [2]. Using a semi-formal language model indicates what the SUT should do and it is the basis for the algorithmic generation of test cases and for the evaluation of the test results. The model itself prescribes the I/O of the SUT. It exactly indicates which input the SUT should accept and what should be the expected response to the input received. The main advantage of MBT is that it allows test automation that goes well beyond the mere automatic execution of manually crafted test cases. It allows the algorithmic generation of large amounts of test cases, including their expected results, completely automatically and correctly from the model of required behavior. The theory allows for the development of sound and complete models.

From an industrial standpoint, MBT shows great potential in efficiently and affordably identifying a higher number of bugs. Currently test automation primarily focuses on executing tests automatically, while the issue of test generation remains unaddressed. Model-based testing seeks to address this by automatically creating comprehensive, sound and complete test suites based on a model, thereby implementing the automated test execution process [3].

From an academic standpoint, MBT is an approach rooted in formal methods that serves as a valuable companion to formal verification and model checking techniques. The ultimate objective of formal verification and

model checking is to establish that a system satisfies predefined properties by proving that a model representing the system indeed fulfills these properties [3]. Therefore, the quality of any verification outcome hinges on the soundness and validity of the underlying model upon which it is built. In contrast, model-based testing takes a different approach by commencing with a (previously verified) model, and its objective is to establish that the real-world implementation of the system behaves in accordance with this model. While in theory the MBT is sound and complete, due to the inherent limitations of actually running the tests MBT can never be complete: testing can only show the presence of errors, not their absence. These limitations include model accuracy, model maintenance, domain expertise, test oracle and automation challenges, which we will elaborate on in the following sections. Practically speaking, it is important to consider these limitations while applying MBT and to complement it with other testing techniques to ensure comprehensive and reliable testing.

2.2 Benefits of MBT

The primary advantage of MBT is its ability to generate test cases, which has been a significant challenge in automation processes. This approach enables the generation of a larger, more extensive, and diverse set of test cases with reduced effort. Furthermore, these test cases are inherently valid as they are constructed based on robust algorithms.

Developing models for MBT offers several advantages, including gaining a deeper understanding of system behavior and requirements and early detection of specification and design errors. Additionally, creating models for MBT leads to an easier implementation for other model-based approaches like model-based analysis, model checking, and simulation. It also forms a natural connection to model-based system development that is becoming an important driving force in the software industry.

2.3 Types and challenges of MBT

There are different kinds of testing, and thus model-based testing, depending on the kind of models being used, the quality aspects being tested, the level of formality involved, the degree of accessibility and observability of the system being tested, and the kind of system being tested [2].

In this thesis we consider MBT as specification-based, formal, black-box, functionality testing of reactive systems. It is testing because the process involves checking specification properties of the BBS. Functionality testing in particular checks whether the system reacts how it should react to the given stimuli. It is being called black-box testing since it compares the externally observable behavior of the system seen as a black box against the previously constructed formal specification (the model). The test is also active since the tester controls the system under test (BBS) by sending stimuli to trigger the SUT and actively observes the responses. The SUT is reactive, meaning that the system will react to external events (stimuli, input, etc) with output events (response, actions, and so on). In such systems the types of outputs are directly connected to types of inputs and also the state that the system is in when receiving the input. Finally, a formal and well-defined theory exists that serves as the basis for model, SUT, and the correctness of SUT in relation to the model. This theory allows for formal reasoning about the completeness and accuracy of generated test suites.

Software has become ubiquitous in our society, with an expanding range of systems relying on it. Whether it's automobiles, airplanes, pacemakers, or refrigerators, software is now responsible for controlling, linking, and overseeing nearly every aspect of these systems. Its presence is vital in ensuring the effective performance and operation of these diverse devices and appliances. Generally, this type of software comprises a vast amount of code, often reaching millions of lines. It involves intricate control flow, complex data structures, and employs distribution and parallelism. The software interfaces are diverse and challenging, catering to a range of functionalities. It oversees various multidisciplinary processes, each with their own complexities. Moreover, these systems are in a constant state of evolution and are integrated into larger systems and systems-of-systems. The components used in these systems can come from a variety of sources, such as third-parties, open-source libraries, or newly developed components.

MBT faces various challenges due to these trends. First, the increasing size of systems makes it impractical to create fully comprehensive models, requiring MBT to handle partial and under-specified models as well as abstractions. In addition, dealing with partial knowledge and uncertainty becomes unavoidable in such scenarios.Secondly, the combination of complicated state-behavior and intricate input and output-data structures must be supported in modeling which is challenging due to the ever increasing complexity. Thirdly, distribution and parallelism imply that MBT must deal with concurrency in models, which introduces additional uncertainty and non-determinism [2]. Furthermore, as complex systems are constructed from subsystems and components, and systems are increasingly integrated into systems-of-systems, model-based testing (MBT) needs to facilitate compositionality. This means enabling the construction of intricate models by combining simpler models together. The ability to compose and integrate these models effectively becomes crucial in addressing the challenges posed by complex and interconnected systems. Finally, the complexity of systems in model-based testing leads to an overwhelming number of potential test cases. Consequently, the key challenge lies in test selection, which involves identifying the tests that can capture the most critical failures within the given constraints of time and budget. Balancing these factors is essential for ensuring efficient and effective model-based testing.

Overall, in order to be applicable for testing modern software systems, model-based testing (MBT) must incorporate several crucial elements. These include supporting partial models, under-specification, abstraction, uncertainty, state and data handling, concurrency, non-determinism, compositionality, and test selection. Despite the existence of numerous academic and commercial MBT tools, the availability of comprehensive tools covering all these aspects is relatively limited.

2.4 The Theory of MBT

A theory for MBT must naturally define first the models that are considered. The modeling formalism determines the kind of properties that can be specified, and consequently the kind of properties for which test cases can be generated. Secondly, it must be precisely defined what it means for an SUT to conform to a model. Conformance can be expressed using an implementation relation, also called conformance relation [4]. Despite the black box nature of

the SUT, it is possible to hypothesize that it can be effectively represented by a model instance within a particular domain of implementation models. This hypothesis is commonly referred to as the testability hypothesis, or test assumption [5]. The testability hypothesis provides a framework for analyzing SUTs by treating them as formal models. It allows for the establishment of a formal relationship between the domain of specification models and the domain of implementation models, known as the implementation relation. The concept of soundness, which ensures that all correct SUTs pass, and exhaustiveness, which verifies that all incorrect SUTs fail, is defined based on this implementation relation.

Model-Based Testing and process algebra are closely related in the context of testing concurrent and distributed systems. Process algebra provides a formal language and mathematical framework for modeling and analyzing concurrent systems. It allows for the precise specification of system behavior, communication patterns, synchronization, and other aspects of concurrent processes. Process algebra formalisms, such as CSP (Communicating sequential processes) [18], CCS (Calculus of communicating systems) [19], and the π-calculus [20] enable the representation and manipulation of processes using algebraic expressions and operators. CSP is an example, suitable for modeling and analyzing concurrent systems. It provides a formal language for describing the behavior of processes and their communication through channels. The CSP model allows us to specify the desired behavior of the system and analyze properties such as safety, liveness, and deadlock-freedom. For example, we can define safety properties to ensure that a buffer is not accessed when it is empty or full. Liveness properties can also be defined to ensure that items are eventually produced and consumed.

Figure 1 indicates a practical example of a CSP model. In this model, *produce*, *consume*, and *buffer* are events representing the actions of producing an *"item"*, "consuming" an item, and accessing the shared "buffer", respectively. The || operator denotes parallel composition, indicating that the processes P, C, and B run concurrently, synchronizing over their common actions.

```
P = (produce -> buffer -> P)
C = (buffer -> consume -> C)
B = (buffer -> B)

System = P || C || B
```

Figure 1.CSP model definition

The use of CSP in this practical example demonstrates how process algebra provides a formal and systematic approach to modeling (and eventually reasoning about) concurrent systems.

In another more complex example, let us consider a scenario where we have a distributed system with multiple components: *Server*, *Clients* and *Load Balancer*. The Clients send requests to the Server, which processes them and sends responses back. *Load Balancer* distributes incoming requests among multiple instances of the Server. Let us model the behavior of this system in CSP. As indicated in Figure 2, we start by defining processes that represent the behavior of each component. Each process can send and receive messages.

```
Process Client(id):
      sendRequest -> receiveResponse -> stop

Process Server(id):
      receiveRequest -> sendResponse -> stop

Process LoadBalancer:
      receiveRequest -> forwardRequest -> stop
```

Figure 2. Defining processes in CSP

In this example, the process *Client(id)* represents the behavior of a Client component identified by *id*. It sends a *sendRequest* message and then receives a *receiveResponse* message. The process *Server(id)* represents the behavior of a Server component identified by *id*. It receives a *receiveRequest* message and then sends back a *sendResponse* message. The process *LoadBalancer* represents the behavior of the load balancer component. It receives a

receiveRequest message, forwards it to one of the Server instances using a *forwardRequest* message, and then stops.

Afterwards as shown in Figure 3, we define the communication channels through which messages are passed between the processes. Then we compose the processes and specify the communication channels to represent the overall behavior of the system.

```
Channel sendRequest, receiveRequest, sendResponse, receiveResponse, forwardResponse

System = LoadBalancer || (Server || Client(1) || Client(2) || Client(3))

System [sendRequest, receiveRequest, sendResponse, receiveResponse, forwardResponse]
```

Figure 3. Defining communication channels and behavior in CSP

In this example, we define five communication channels: *sendRequest*, *receiveRequest*, *sendResponse*, *receiveResponse*, and *forwardRequest*. These channels facilitate the communication and synchronization between the components. The composed process *System* represents the behavior of the entire distributed system. It is created by composing the *LoadBalancer*, *Server*, and multiple *Client* processes using the parallel composition operator ||. The square brackets [*sendRequest*, *receiveRequest*, *sendResponse*, *receiveResponse*, *forwardRequest*] specify the communication channels used by the processes. With the processes and communication defined, we can analyze the behavior of the system. This may involve checking for desired properties, exploring different scenarios, or generating test cases based on the model.

One of the specification formalisms studied in the realm of formal conformance testing is that of labeled transition systems. A labeled transition system is a structure consisting of states with transitions, labeled with actions, between them. The formalism of labeled transition systems can be used for modeling the behavior of processes, and it serves as a semantic model for various formal specification languages including CSP. Testing theory and algorithms for the generation of tests from labeled transition system specifications have been developed during the last few decades [7,8]. All these methods, as most of the theory on labeled transition systems, are based on synchronous, symmetric communication between different processes: communication between two processes occurs if both processes offer to

interact on a particular action, and if the interaction takes place it occurs synchronously in both participating processes, without a notion of distinction between input and output actions. For testing theories a particular case where such communication occurs is the modeling of the interaction between a tester and an implementation under test during test execution [4].

A labeled transition system (LTS) [3] is a 4-tuple (S, L, T, s0) where, S is a countable, non-empty set of states, L is a countable set of labels, $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation and $s0 \in S$ is the initial state [4].

To illustrate the concepts of (LTS) we represent the behavior of a simple vending machine. The LTS captures the different states of the vending machine and the transitions between those states in response to various events or actions. Let us imagine a simple vending machine with the set of states S = {Idle, Selecting, Dispensing, Refilling, OutOfStock}, the set of actions or events L = {InsertCoin, SelectProduct, DispenseProduct, RefillMachine, ProductOutOfStock}, and the set of possible transitions T = {(Idle, InsertCoin, Selecting), (Selecting, SelectProduct, Selecting), (Selecting, DispenseProduct, Dispensing), (Dispensing, DispenseProduct, Idle), (Idle, RefillMachine, Refilling), (Refilling, RefillMachine, Idle), (Selecting, ProductOutOfStock, OutOfStock)}. The initial stage for this vending machine is Idle.

We can illustrate an LTS using a graph with nodes representing states and edges representing transitions. For example the graph representing the vending machine LTS described above is shown in Figure 4.

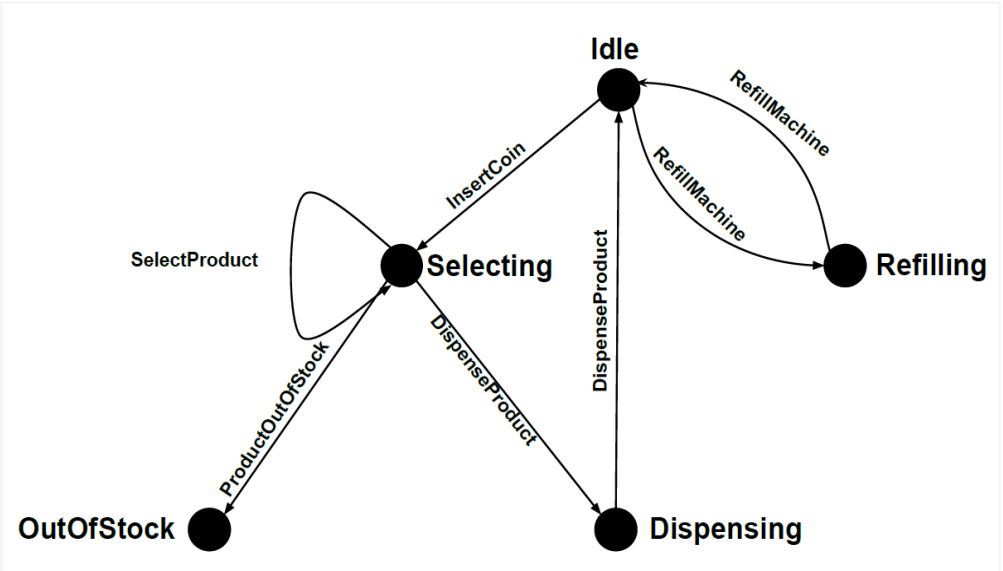

Figure 4. LTS illustration of a simple vending machine

Starting from the initial state *Idle*, the vending machine can transition to the *Selecting* state when a user inserts a coin performing *InsertCoin*. From the *Selecting* state, the user can either change their product selection while staying in the same state executing *SelectProduct* or confirm their selection, leading to the dispensing of the product by executing *DispenseProduct* and thus transitioning to the *Dispensing* state. If the vending machine runs out of stock for a particular product while a user is in the selecting state, it transitions to the *OutOfStock* state with action *ProductOutOfStock*. The *OutOfStock* state indicates that the vending machine is currently out of stock for that specific product. The vending machine can also transition to the *Refilling* state if someone refills it with products by offering *RefillMachine*. Once the refilling is complete, the vending machine transitions back to the *Idle* state with the same (synchronized) event *RefillMachine*.

A similar approach can be used for capturing behavior of a Bulletin Board System to be able to validate the conformance level of the system using a model which we will explore more in following chapters.

On the practical side TorXakis is an advanced software solution that incorporates a similar approach wherein system specifications can be precisely described using a process algebra thus obtaining a model. This model can be used to algorithmically generate test cases and apply them against the SUT.

TorXakis is one of the MBT tools, we have chosen it for no particular reason. There are other MBT tools available such as Modbat [14] which is an open-source tool, and TOSCA [15] which is a commercial tool. Another popular form of verification model checking, which uses logical rather than algebraic specifications. Model checking is popular in the industry, with the most popular tools including SPIN [16] and NuSMV [17].

2.5 Bulletin Board Systems (BBS)

A Bulletin Board System (BBS) is a very old computer-based platform that facilitates communication, file access, and discussion among users. Typically hosted on a server, a BBS allows users to connect through dial-up connections or internet protocols like telnet. Users interact with the BBS using terminal software, which provides a text-based interface. It rose in

popularity during the 1980s and 1990s but declined with the emergence of the internet. BBS offered features like message boards, file libraries, user profiles, and online games, serving as virtual gathering spots for individuals sharing common interests. While their prevalence has diminished, specialized versions of BBSs still persist today.

The best known early BBS is a Chicago-based community started in 1978 during a legendary blizzard by Ward Christensen and Randy Suess. Their innovation was to create a custom interface that would detect an incoming call on the modem and *"cold-boot"* the caller directly into the special host program, where they could then read articles and leave a message [6].

The BBS we are intending to grade is a simplified version of a real BBS. It was given as a final project in a network programming course at Bishop's University in the 2021-2022 academic year. In addition to basic interaction with clients, the specification also requires the ability to run multiple instances of the server which synchronize with each other using the two-phase commit protocol. The specification is given in the following chapter.

In this thesis we are trying to grade 30 submissions based on their ability on basic client-server communication, which includes performing READ, WRITE, REPLACE and other available commands on the BBS and also basic peer synchronization. We will then continue by testing replication to evaluate the implementation of the two-phase commit protocol. We will discuss the grading process in Chapters 3 and 4.

Chapter 3:

Preliminaries

When performing a formal, specification based testing there are various concepts and objects that we need to pay attention to which we explore in this section. These concepts and objects result in a framework for formal testing with respect to a formal specification of its functional behavior which we can observe at a high level of abstraction in Figure 5 [3].

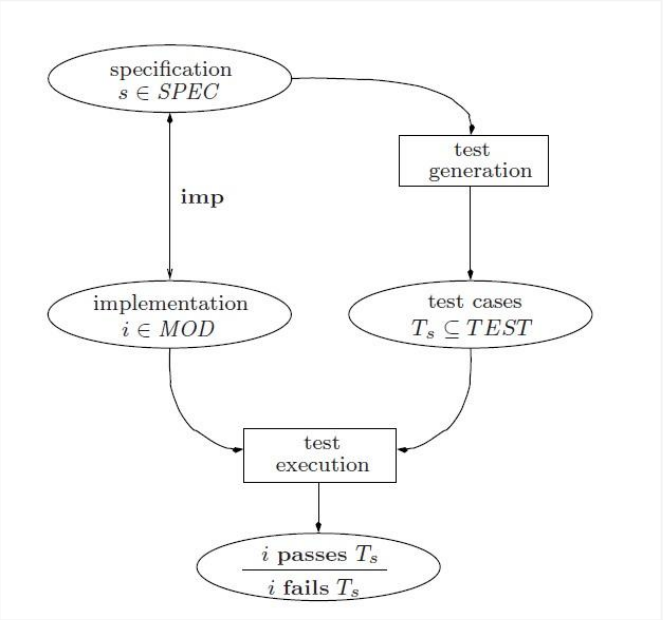

Figure 5. The formal, specification based testing process

The very first step in testing is having a system to perform tests on which as we explained earlier is tha *system under test* or SUT. As SUT can be a real world object like a piece of hardware, a software with all the modules, an embedded system consisting of software embedded in some physical device, or a process control system consisting of sensors and actuators. Since model based testing is a black box testing, we are treating SUT as a black box

exhibiting behaviors and interacting with its environment, but without any knowledge about its internal processes and structure. This means that the tester will only interact with the system interface and the aim of this test is to validate the behavior of the SUT.

The correctness of an SUT is expressed as conformance to a *specification*. Basically a specification is what the SUT should do and what it should not do, and in this thesis we call it a model of a system's expected behavior. In formal testing the specification is expressed in a formal language with formal syntax and semantics, such as a process algebra. By means of testing we want to check if the I/O behavior of the SUT conforms to this specification.

To check if the SUT conforms to the specification we need a formal definition of *conformance*. This definition should relate the SUT with the specification. In this thesis we are generating test case models for different scenarios and then we validate the conformance of the SUT with all the created models

The behavior of the SUT will be investigated by performing experiments on it, or *testing*. Each experiment in general consists of sending commands as stimuli to the SUT and validating the responses. The specification of such an experiment, including both command and expected responses, is called a *test case*. The process of applying a test to a SUT is called *test execution*. Test execution can either be successful which means that the observed response corresponds to the expected response or can be unsuccessful which means the observed response does not match the expected response. The successful execution of a test is referred to as a pass and unsuccessful execution of a test is a fail.

Finally, the final score is given to the model by comparing model generated results and correct answers. Then we can see the possibility of automation using model-based testing for our system under test. The subsequent sections will examine the comprehensive process of testing, covering the initial stages all the way through to its conclusion.

3.1 System Under Test (SUT)

The very first step in testing is having a system to perform tests on which as we explained before is deemed the system under test or SUT. Since model based testing is a black box testing technique, we are treating SUT as a

black box exhibiting behaviors and interacting with its environment, but without any knowledge about its internal processes and structure. It means the tester will only interact with the system interface and the aim of this test is to validate the behavior of the SUT.

In this contribution the SUT is a simple execution of a Bulletin Board Server (BBS) which features client-server communication, database storage, and replication. All the executions have been implemented by Master's students at Bishop's University. There are a total number of 30 submissions available. In the upcoming discussion, we will outline the fundamental rules that must be adhered to for all implementations.

The testing process can pose significant challenges due to the presence of buggy implementations in the SUT, which have not been rectified or verified prior to testing. These are coursework submissions, not polished, commercial products, and they should be graded using a more nuanced approach that simply pass/fail. This presents a challenge for our model, as in principle complete adherence to the rules is required to validate an implementation using the model, yet each student may employ their own approach to implement the system and thus introduces different kinds of deviations from the given specification (some minor, some not so minor). The thesis supervisor of this student has observed in the past numerous times that students tend to not appreciate the need to observe the given application protocol in a network application, and often feel the urge to introduce their own idiosyncratic deviations.

3.1.1 Implementation

Implementations should represent a simple BBS. The server accepts one-line messages from multiple clients, stores them in a local file, and serves them back on request. The name of the file is given by a command line parameter and is hereby referred to as the *bbfile*. Messages are identified upon storage by an unique number established by the server, and by the "sender" of the message.

3.1.2 Application Protocol

While there may not be strict enforcement regarding the implementation approach or method to be used, there are nonetheless strict protocols that must be adhered to which will be outlined in this section. This is referred to as the application protocol of the client-server application.

Every command and response consists of a single line of text. The server should handle any combination of the characters '\n' and '\r' as line terminator and should send back responses terminated by a single '\n'. Implementations should be able to be tested using telnet as a client or indeed any other client capable of sending and receiving plain text. Each server response should contain a status which will be used for validation.

1. Greeting: At the beginning of the interaction the server sends the Greeting message to the client that just connected. If the connection is established successfully then a status *"0.0"* should be sent, followed by a one-line "greeting" that is, some (possibly empty) message intended for human consumption. There is no particular format for the greeting text, but it is strongly suggested for this text to summarize the commands available to clients.
2. The command "USER *name*" should establish the user name of all the subsequent messages being posted by the respective client. The argument name is a string not containing the character /. Future messages posted by the respective client will be identified as posted by *name*. Normally, a client will send this command at the beginning of the session, but the server should handle the case in which this command is sent more than once during the interaction with a particular client, as well as the case when a client does not send a "USER" command at all (case in which the poster will be "nobody"). The server response if the command executes successfully should be "1.0 Hello *name text*" where *text* is some (possibly empty) message intended for human consumption. Whenever the user *name* contains unacceptable characters (including but not necessarily limited to '/') or is otherwise incorrect the response of the server must be "1.2 ERROR USER *text*" where *text* is once more intended for human consumption and explains the issue encountered in processing the request.
3. The command "READ *message-number*" asks for the message number *message-number*. In the event that this message exists on the

bulletin-board, the server will send in response one line of the form “2.0 MESSAGE *message-number poster/message*” where *message* represents the requested message, prefixed by its *poster* (as identified by the USER command in effect at the time of posting). If a message with *message-number* does not exist, then the server must send back the response “2.1 UNKNOWN *message-number text*”, where *text* is a message for human consumption. If the server encountered an internal error while serving the request (e.g., the unavailability of the bulletin board file), then the response should look like “2.2 ERROR READ *text*”, where again, *text* is an explanatory message with no particular structure.

4. The command “WRITE *message*” sends a message to the server for storage. The server will store the message into the bulletin board file as a line of the form “*message-number*/*poster*/*message*”, where *message-number* is a unique number assigned by the server, and *poster* is the poster of the message as specified by a previous "USER" command issued by the respective client (*nobody* if no USER command has been issued by that client). Upon successful storage, the server returns the message “3.0 WROTE message-number”. When an error occurs during the storage process, the server responds should instead look like “3.2 ERROR WRITE *text*”. The receipt of such a response must guarantee that no message has been written to the bulletin board file.
5. The command “REPLACE *message-number/message*” asks the server to erase the message number *message-number* and replace it with *message* (which will be assigned the same message number as the original). The poster is also changed to the current poster as identified by a previous *"USER"* command (*nobody* if no such command has been issued). The server response is identical to the response to a ″WRITE″ request. Additionally, when message number *message-number* does not exist in the bulletin board file (case in which no message is added to the file) then the response should be “3.1 UNKNOWN *message-number*”.
6. The command “QUIT *text*” signals the end of interaction. Upon receipt of this message the server sends back the line “4.0 BYE *some-text*” and closes the socket. The same response (including the socket close) is given by the server to a client that just shuts down its connection. The server should always close the socket in a civilized manner by shutting down the socket before closing it.

3.1.3 Other implementation requirements

The server must be robust, in the sense that no message shall be lost when the server is terminated, except possibly a message that is currently being written to disk. The bulletin board file (where messages will be saved) should be considered too large to be kept completely in memory. The server must also be efficient, in the sense that it must not rewrite the whole bulletin board file upon the receipt of each and every message. It should use the file system as little as possible (within the robustness requirements above).

3.1.4 The Bulletin Board File

The bulletin board file must be created to store messages from *"WRITE"* requests. It specified in the configuration file or on the command line must be created if nonexistent and must be re-used as is otherwise (rather than being overwritten). Message numbers are assigned by the server. No two messages can have the same number in any bulletin board file. In particular, if the server is started on an existing file it should inspect the file on startup and make sure that any new message written to the file has an associated number that does not conflict with existing message numbers.

3.2 Specification (The model)

The model is written in the TorXakis modeling language. TaXakis uses its own process-algebraic language Txs (pronounced t`ex`es) to express models. The language is strongly inspired by the process-algebraic language Lotos, and incorporates ideas from Extended Lotos and mCRL2, combined with plain state-transition systems. The semantics is based on STS, which in turn has an LTS semantics. Having its roots in process algebra, the language is compositional. It has several operators to combine transition systems: sequencing, choice, parallel composition with and without communication, interrupt, disable, and abstraction. Communication between processes can be multi-way, and actions can be built using multiple labels.

A (collection of) model file(s) contains all the definitions necessary for expressing the model: channels, data types, functions, constants, and process definitions, which are all combined in a model definition. In addition, the

model file contains some testing specific aspects: connections as well as encodings/decodings. A connection definition defines how TorXakis is connected to the SUT by specifying the binding of abstract model channels to concrete sockets. Encodings/decodings specify the mapping of abstract messages (ADTs) to strings and vice versa [2].

In this section, we will provide a description of our model and cover all of its functions in detail. Overall, our model consists of six sections, TYPEDEF, FUNCDEF, MODELDEF, CHANDEF and CNECTDEF.

TYPEDEF defines data types. Our model includes two different data types. First, the *Server_response* data type basically categorizes the server responses. The second type is *Message_list* which is a list to save the messages that we send to the server. It acts as a local (and incomplete) copy of the bulletin board file and is used to check the messages as we read them back.

```
TYPEDEF Server_Response ::=
        No_response
           | Write     {wstatus :: String; write_number :: String}
           | Read      {rstatus :: String; read_text :: String}
           | Error     {estatus :: String; error_text :: String}
           | User      {ustatus :: String; username :: String}
           | Greet     {gstatus :: String; greeting_text :: String}
           | Quit      {qstatus :: String; bye_text :: String}
ENDDEF

TYPEDEF Message_list ::=
        No_message
           | Messages  {mstatus :: String; message_num :: String; poster :: String;
message :: String; rest :: Message_list}
ENDDEF
```

Figure 6. Defined data typed in Txs

FUNCDEF defines functions in TorXakis. Our model contains an overall of 14 functions. Using these functions we are able to define specifications and confirm the validity of the SUT. To define a function in TorXakis modeling language we proceed as shown in Figure 7.

```
FUNCDEF name ( arg :: type ) :: return_type ::=
       Body of function
ENDDEF
```

Figure 7. How to define functions in Txs

The following are the functions used in our model.

FUNCDEF Valid_greeting: This function is used for validating greeting messages. This function looks for *"0.0"* as greeting status plus *"greeting"* as the start of the greeting message.

FUNDEF Valid_user: This function is used for validating the response we receive back from the server after sending a *"USER username"* command.

FUNCDEF Valid_read: This is similar to the previous functions, but for validating the response to the *READ message-number* command.

FUNCDEF Substring: This function is used to retrieve a particular section from a longer string.

FUNCDEF Message_exist: This function is used to verify the presence of a specific message in memory, enabling the retrieval of the message or the substitution with a new message.

FUNCDEF Before_slash and *"FUNCDEF After_slash"*: These functions are used to extract text from various server responses, specifically around the forward slash ("/") character.

FUNCDEF Find_message: This function is used to look for a particular message within a list of messages and retrieve the match (if any).

FUNCDEF Add_message: This function is used for adding a new message into the message list.

FUNCDEF Edit_message: This function is primarily used for testing the REPLACE command, and involves substituting a new message with an existing message already stored in the message list.

FUNCDEF Last_message: This function is used for generating the read commands like *"*READ *message-number"*. It will combine the string *"*READ*"* with the message number of the last message.

FUNCDEF Select_replace: This function is similar to the previous function and generates a REPLACE command as an input by combining the string *"*REPLACE*"*, a message number, the forward slash character "/", and the new replacement message.

FUNCDEF Valid_quit: This function is used for validating the response that is received back from the server for the *"*QUIT *text"* command.

MODELDEF defines a model, with its input and output channels for external communication, and the definition of its behavior. Figure 8 illustrates a model with one input and one output channel, together with its behavior definition expressed as a process (to be explained below).

```
MODELDEF Model ::=

       CHAN IN       Command
       CHAN OUT      Response
       BEHAVIOUR     write_first[Command, Response](No_messages, "USER mo",
"WRITE first")

ENDDEF
```

Figure 8. Defined model's in/out and behavior in Txs

CHANDEF defines all the channels that are used on the highest level in the TorXakis code, i.e., in model definitions (MODELDEF) and in connection definitions (CNECTDEF). For each channel the types of messages communicated via that channel are defined. At the CHANDEF level, channels do not have an I/O direction yet; instead, the I/O behavior is specified at the level of MODELDEF and CNECTDEF. In Figure 9 CHANDEF defines two channels: Command, and Response, with messages of types String.

```
CHANDEF Chans ::=
      Command  :: String
    ; Response :: String
ENDDEF
```

Figure 9. Defined channels in Txs

CNECTDEF defines connections with the outside world, and the mapping from abstract TorXakis channels to the concrete outside-world connections. Currently only socket connections (of type String) are supported. A socket has a hostname, such as *"localhost"*, and a port number. Figure 10 specifies the socket connection (on localhost and port number 9550), called Sut where TorXakis is the client side. There is one outgoing and one incoming channel Command and Response, respectively. *"ENCODE"* refers to the process of transforming data or a message from its internal representation within a component into a format suitable for communication over a channel. It involves converting the data or message into a serialized or encoded form that can be sent through the communication channel.

Similarly, *"DECODE"* refers to the process of transforming received data or a message from its encoded form, back into the internal representation within a component. It involves extracting the relevant information from the received data and converting it back to its original format or representation. The *"ENCODE"* and *"DECODE"* actions allow us to specify these transformations explicitly outside the model definition. Alternatively, the conversion from string representation to internal data can be done within the model definition. This is our approach and therefore in our case the encoding and decoding procedures just pass a string along unmodified.

```
CNECTDEF Sut ::=
       CLIENTSOCK

       CHAN OUT      Command                    HOST "localhost" PORT 9550
       ENCODE        Command  ? c           -> ! c

       CHAN IN       Response                   HOST "localhost" PORT 9550
       DECODE        Response ! p           <- ? p

ENDDEF
```

Figure 10. Defined connections in Txs

PROCDEF is where we define our processes that model the behavior of the SUT. Our model consists of 6 process definitions, each for testing an aspect of the BBS. Figure 11 indicates all process definitions that have been implemented in our model to validate various functionalities.

```
PROCDEF write_first [Com::String; Res::String](l::Message_list; u::String; f::String)::=
        Res ? r [[Valid_greeting(r)]]
        >-> Com ! u
        >-> Res ? r [[Valid_user(r)]] >-> Com ! f
        >-> Res ? r [[Valid_write(r)]] >->read_first[Com,Res](Add_message(r, u, f, l), u)
ENDDEF

PROCDEF read_first [Com::String; Res::String](l::Message_list; u::String)::=
        Com! Last_message (1)
        >-> Res ? r [[Valid_read(r)]] >-> write_second [Com,Res] (l,u,"WRITE second")
ENDDEF

PROCDEF write_second [Com::String; Res::String](l::Message_list; u::String; s::String)::=
        Com ! s
        >-> Res ? r [[Valid_write(r)]] >->read_second [Com,Res] (Add_message(r,u,s,l),u)
ENDDEF

PROCDEF read_second [Com::String; Res::String] (l::Message_list; u::String)::=
        Com ! Last_message(l)
        >-> Res ? r [[Valid_read(r)]]>-> replace[Com,Res](l,u,"replaced message")
ENDDEF

PROCDEF replace [Com::String; Res::String](1::Message_list; u::String; h::String) ::=
        Com ! Select_replace(l, h)
        >-> Res ? r [[Valid_write(r)]]>-> quit [Com,Res] (l, u, "QUIT bye")
ENDDEF
PROCDEF quit [Com::String; Res::String](l::Message_list; u::String; q::String) ::=
        Com ! q
        >-> Res ? r [[Valid_quit(r)]]>-> EXIT
ENDDEF
```

Figure 11. Defined processes in Txs

3.3 Conformance

In model-based testing (MBT) conformance refers to the degree to which the implementation of a system under test (SUT) adheres to its specified model or requirements. Conformance testing is conducted to verify that the SUT behaves correctly and conforms to the expected behavior defined by the

model. The process of conformance testing typically involves comparing the actual behavior of the SUT with the expected behavior derived from the model. This comparison can be done by executing test cases derived from the model on the SUT and observing its responses. The observed behavior is then compared against the expected behavior specified by the model or requirements.

The degree of conformance is determined by the level of agreement between the observed behavior of the SUT and the expected behavior defined by the model. If the observed behavior closely matches the expected behavior, the SUT is considered to have a high degree of conformance. On the other hand, if significant discrepancies or failures are observed, the SUT has a lower degree of conformance, indicating issues or non-compliance.

Conformance testing is obviously an important part of the overall MBT process, as it provides confidence that the implemented system adheres to its specified model or requirements. It helps identify areas where the SUT may require improvements, adjustments, or further testing to achieve the desired level of conformance.

In this thesis we are testing basic client-server communication as well as the creation, synchronization, and usage of the bulletin board file, which basically covers all the commands (USER, WRITE, READ, REPLACE, and QUIT). The level of conformance can be described as how our BBS are behaving compared to the expected behavior which is described in the previous section.

Chapter 4:

Grading

Once we have a system under test (SUT) and a model (which is the combination of all the definitions that we have mentioned in Section 3.2), we can use TorXakis for test execution. Note that TorXakis is based on ioco testing [13], which defines an algorithmic approach to verification. The model (specification) is described using a process algebraic language as described earlier, and then the process of test generation and application is fully automated by the tool. At the end of the process TorXakis provides an overall verdict for all the tests derived from the model and applied to the SUT, which can either be *pass* or *fail*.

4.1 Execution of tests

As discussed in Section 3.2, a TorXakis model is a collection of different kinds of definitions. TYPEDEF defines various types of data. FUNCTDEF defines functions. MODELDEF defines the models inputs and outputs for external communication, as well as the desired behavior. Finally, CHANDEF defines the channels with their typed messages. TorXakis assumes that an SUT communicates by receiving and sending typed messages. A message received by the SUT is an input and thus in our case an action initiated by the user [2]. A message sent by SUT is an SUT output, and it is observed and validated by the tester (TorXakis).

At this point we start the main testing process. We apply model-based testing using TorXakis to test the BBS implementations. We are testing basic client-server communication as well as the creation, synchronization, and usage of the bulletin board file, which basically covers the commands USER, WRITE, READ, REPLACE, and QUIT.

For our SUT there is one input channel and one output channel, *command* and *response*, respectively, where both channels have the type *String* as shown in Figure 9 (Section 3.2). We have benefited from a virtual Linux server to perform the testing process. For each BBS we first start the

BBS on the localhost and port number 9550 with the command *"./bbserv -b bbfile1 -f -T 5 -p 9550"* as shown in Figure 12.

```
<mabdolahi@linux: 1-Passed/aarigela-anil > ./bbserv -b bbfilel -f -T 3 -p 9550
process ID of current process : 438413
bulletin server up and listening on port 9550
Synchronization server up and listening on port 10000
```

Figure 12. How to launching BBS on localhost:9550

Then we start TorXakis and launch our model through TorXakis. Afterwards, we start the tester inside TorXakis command line with the command *"tester Model Sut"* which initiates the process and starts running our model as shown in Figure 13. Then by sending the *"test 20"* command we initiate the testing process for 20 steps which is an empirical value achieved by experimenting, and in our experience 20 steps is enough to pass all the tests completely. On line 1, TorXakis receives an output from the SUT on channel *Response*, which indicates the greeting message with the correct status *"0.0"*. Afterwards, TorXakis generates inputs to the SUT such as on line 2 indicating that on channel *Command* an input command *USER* with the username value of *"mo"* has occurred. The input has been sent to the SUT by TorXakis according to the model. This action is followed, on line 3, by an output from the SUT on channel *Response*, which is the proper response for the *USER* command with the correct status *"1.0"*. Then TorXakis checks that this is indeed the expected response. The process will continue on line 4 by sending the *"WRITE first"* command on channel *Command* to the SUT. For this specific BBS submission the write command is followed by the *"3.0 WROTE 43"* output on channel *Response*, which means that the write process has successfully completed and the message *"first"* should exist in the bulletin board file at this point.

Then on line 6 TorXakis will send the command *"READ 43"* on channel *Command* to test the functionality of the BBS. Then the SUT sends back the response *"2.0 43 mo/first"* on channel *Response*, which means the *READ* command has been successfully completed since the status *"2.0"* is a validation for completion of this command. Our model also validates the message using the message number and will make sure that the specific message with its message number exists in the model's message list. The

next two sets of commands and responses are similar to the previous *WRITE* and *READ* commands. Afterwards, as shown on line 12, TorXakis will send the *"REPLACE 44/replaced message"* command on channel *Command* to the SUT, and then as shown on line 13 the response *"3.0 WROTE 44"* is received on channel *Response*, which is indicating the correct behavior of the SUT in this situation. Next, as indicated on line 14, TorXakis will send a *READ* command to verify the previous *REPLACE* command which in this case is followed by the proper response from the SUT. Finally on line 16 TorXakis sends the command "QUIT bye" through the *Command* channel, signaling the termination of the client-server communication. The specific Bulletin Board System (BBS) being tested successfully responds to this command appropriately.

```
< mabdolahi@linux:1-Passed/aarigela-anil > torxakis Model.txs

TXS >>  TorXakis :: Model-Based Testing

TXS >>  txsserver starting: "localhost" : 41563
TXS >>  Solver "z3" initialized : Z3 [4.12.1 - build hashcode 3012293c35eadbfd73e5b94adbe50b0cc44ffb83]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["Model.txs"]
TXS >> tester Model Sut
TXS >>  Tester started
TXS >> test 20
TXS >>  .....1: OUT: Act { { ( Response, [ "0.0 Welcome to Bulletin Server" ] ) } }
TXS >>  .....2: IN: Act { { ( Command, [ "USER mo" ] ) } }
TXS >>  .....3: OUT: Act { { ( Response, [ "1.0 Hello mo welcome back" ] ) } }
TXS >>  .....4: IN: Act { { ( Command, [ "WRITE first" ] ) } }
TXS >>  .....5: OUT: Act { { ( Response, [ "3.0 WROTE 43" ] ) } }
TXS >>  .....6: IN: Act { { ( Command, [ "READ 43" ] ) } }
TXS >>  .....7: OUT: Act { { ( Response, [ "2.0 43 mo/first" ] ) } }
TXS >>  .....8: IN: Act { { ( Command, [ "WRITE second" ] ) } }
TXS >>  .....9: OUT: Act { { ( Response, [ "3.0 WROTE 44" ] ) } }
TXS >>  ....10: IN: Act { { ( Command, [ "READ 44" ] ) } }
TXS >>  ....11: OUT: Act { { ( Response, [ "2.0 44 mo/second" ] ) } }
TXS >>  ....12: IN: Act { { ( Command, [ "REPLACE 44/replaced message" ] ) } }
TXS >>  ....13: OUT: Act { { ( Response, [ "3.0 WROTE 44" ] ) } }
TXS >>  ....14: IN: Act { { ( Command, [ "READ 44" ] ) } }
TXS >>  ....15: OUT: Act { { ( Response, [ "2.0 44 mo/replaced message" ] ) } }
TXS >>  ....16: IN: Act { { ( Command, [ "QUIT bye" ] ) } }
TXS >>  ....17: OUT: Act { { ( Response, [ "4.0 BYE mo" ] ) } }
TXS >>  ....18: OUT: No Output (Quiescence)
TXS >>  no more actions
TXS >>  PASS
TXS >>
```

Figure 13. Torxakis test run (PASS)

As shown in Figure 13 the tested BBS has passed the testing successfully without any error or minor failure. Here is now an example of a submission that fails the test. Similarly to the previous example, the testing process begins by receiving the greeting message on the *Response* channel. This

message with the status number *"0.0"* is a valid response in this particular case as well. TorXakis then starts the next part of the testing process by sending the *"USER mo"* command on input channel *Command*, and received the response from the SUT on output channel *Response,* again as specified. Afterward, TorXakis attempts to write a message to the bulletin board file by sending the command *"WRITE first"* through the *Command* input channel. However, in this case, the System Under Test (SUT) does not send back the proper response which should be of form *"3.0 WROTE message_niumber"*. This behavior leads to the respective submission failing the test as shown in Figure 14.

```
< mabdolahi@linux:2-Failed/pgokhru-Project1 > torxakis Model.txs

TXS >>  TorXakis :: Model-Based Testing

TXS >>  txsserver starting: "localhost" : 44125
TXS >>  Solver "z3" initialized : Z3 [4.12.1 - build hashcode 3012293c35eadbfd73e5b94adbe50b0cc44ffb83]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["Model.txs"]
TXS >> tester Model Sut
TXS >>  Tester started
TXS >> test 20
TXS >>  .....1: OUT: Act { { ( Response, [ "0.0 Welcome!! Greetings from BBServer" ] ) } }
TXS >>  .....2: IN: Act { { ( Command, [ "USER mo" ] ) } }
TXS >>  .....3: OUT: Act { { ( Response, [ "1.0 HELLO mo, Hope you're doing well!!" ] ) } }
TXS >>  .....4: IN: Act { { ( Command, [ "WRITE first" ] ) } }
TXS >>  .....5: OUT: No Output (Quiescence)
TXS >>  Expected:
TXS >>  [ ( { Response[ $"Response"$1225$4$1$ ] }, [], Valid_write( $"Response"$1225$4$1$ ) ) ]
TXS >>  FAIL: No Output (Quiescence)
TXS >>
```

Figure 14. TorXakis test run (FAIL)

Similar testing has been performed on 28 other BBS implementations. Figures 12 and 13 indicate the grades achieved using MBT against grades assigned using a human supervised method for all the 30 submissions. The human supervised testing is what was used to mark the submission in the first place and consists of connecting to the server using telnet, issuing appropriate commands, and observing the output.

In our assessment we used the following grading criteria:

Successful establishment of the client-server connection	Achieves 1 mark
Receive of greeting and completion of USER command	Achieves 1 mark
Completion and validation of WRITE command	Achieves 1 mark
Completion of READ command and verification of the BBfile	Achieves 1 mark
Completion of REPLACE command	Achieves 1 mark
Proper ending of the connection	Achieves 1 mark
Any minor issue	Deducts 0.5 marks

A simple side by side comparison between Figure 12 and Figure 13 shows that TorXakis grading was slightly harsher than the human supervised method since it is more precise and will not overlook any minor issues. A more detailed exploration of the achieved results will be offered in Chapter 5.

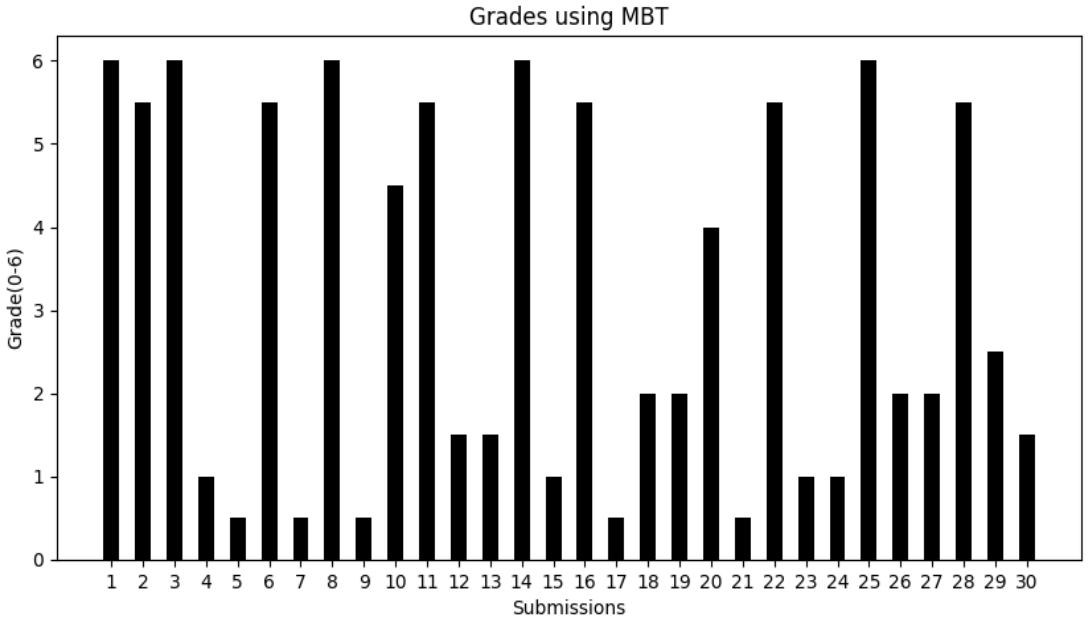

Figure 12. Grades achieved using MBT

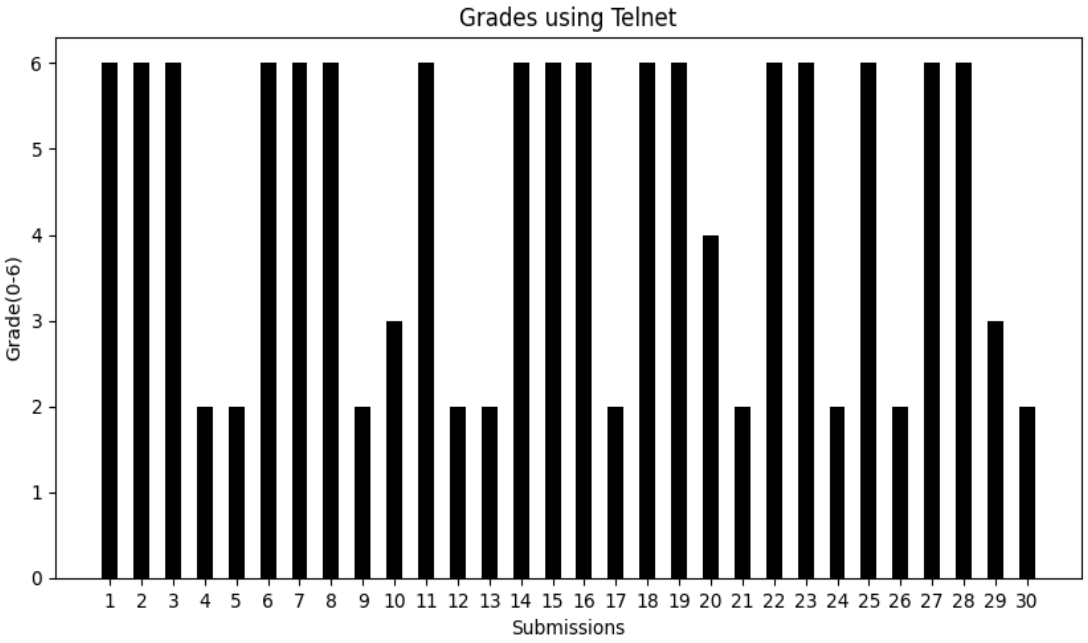

Figure 13. Grades achieved using human supervised method (telnet)

4.2 Challenges

During the testing processes, we encountered numerous challenges. Despite having instructions to follow, we faced minor issues that could potentially disrupt the verification of TorXakis and the model's output. These issues, referred to as minor errors, do not necessarily alter the outcome or purpose of the SUT, but they do require careful observation.

The first challenge arises during the modeling step. Although TorXakis is one of the best and complete tools for MBT, the language still suffers from lack of proper documentation and user manual. It needs a certain level of experience and expertise to be able to build a robust and reliable model. Without having a complete model, the results cannot be trusted or as we mentioned earlier the level of conformance would not be sufficient.

Secondly, one of the most common minor errors we encountered was related to alphabetic errors. These errors likely arose due to a lack of concentration and observation during the implementation phase. In some cases, a successfully completed process would return an incorrect status. We had to continuously assess whether such typos should affect the grade, and adjust our model accordingly.

On a related note, some SUTs occasionally failed to send the appropriate status at the beginning of the response. However, the response

itself was actually complete and valid. In most cases, this issue cannot be disregarded as it would be a major concern. However, if this problem only occurred in the initial greeting message, we considered it a minor issue and utilized the model to proceed with the rest of the testing, which most of these submissions pass. This kind of bugs and errors are expected in student projects, so we made sure to thoroughly test most submissions after addressing these minor issues in the model.

4.3 Synchronization

Testing synchronization involves creating multiple instances of SUT to simulate a distributed environment. This process allows us to test the system's behavior and interactions in multiple instances. In this thesis this test was most of the time a failure since most submissions were buggy and could not synchronize properly over multiple instances, which is the main point of this test. In this section we are briefly explaining the idea behind the synchronization test and our limited success on the matter.

Set up multiple instances: We start by creating three instances of the bulletin board system on localhost. Each instance will run on a different localhost port. For example, we could have three instances running on ports 8000, 8001, and 8002. These instances are supposed to synchronize with each other on each WRITE or REPLACE command using a two-phase commit protocol such that their bulletin board files are identical at all times. READ requests on the other hand are processed locally by the respective instance.

Connect to each instance: Once the instances are running, our model connects to two instances simulating different clients. Each client can establish a connection with a specific instance by specifying the corresponding localhost port. For example, one client might connect to port 8000 and another to port 8001. Note that in this case our TorXakis model uses four channels (two for each connection).

Perform actions: With the clients connected, we can now simulate various actions on the SUT. These actions can include writing messages, reading messages or any other operations supported by the system. For example, one client can write a message to one instance by sending the WRITE command. The SUT should ensure that the command is replicated and synchronized across all instances.

Validate consistency: After performing actions on the SUT, we can verify the consistency of the replicated data. This involves checking if the written messages are visible and accessible from all instances. To validate consistency, we can connect to different instances as clients and attempt to read the messages written by other clients on different instances. If the system is correctly synchronized, we should be able to read the same messages from any instance.

Testing synchronization helps uncover potential issues in a distributed system, such as data inconsistency and synchronization problems. By simulating multiple instances, you can assess the system's resilience, fault tolerance, and overall performance.

Testing the submissions with TorXakis was disappointing, as only three submissions passed. We will comment in the next chapter on this failure.

Chapter 5:

Discussion and conclusion

In the previous chapter we demonstrated that grading using MBT and TorXakis as a test platform would be possible and could even be preferred to conventional human supervised grading methods as long as the system under test is implemented based on strict criteria which allows the machine to be able to verify and grade the output. Now we are going to indicate and compare results achieved with TorXakis and MBT with results achieved using human supervised methods.

Ideally, to be able to grade the SUT using MBT and TorXakis we need to have machine readable parts included in responses that we receive from the SUT. Machine readable sections provide a structured format that ensures consistency and accuracy in the responses. By having a well-defined syntax or format, we can ensure that the responses are interpreted correctly by TorXakis. In this project the machine readable part is the status number that the SUT is supposed to send back at the start of every response. If this part is missing we can still process the response but human supervision is needed to be able to make a solid verdict.

On the other hand, by comparing results achieved by both methods we confirmed that they are similar to a certain degree. Figure 14. We can see submissions that passed the testing with a perfect grade on the human supervision method but either failed on MBT or passed with a grade usually lower than what they have achieved in the human supervised method. This makes complete sense since TorXakis is a machine and it basically will not overlook minor issues but human supervised methods can be affected to a certain degree.

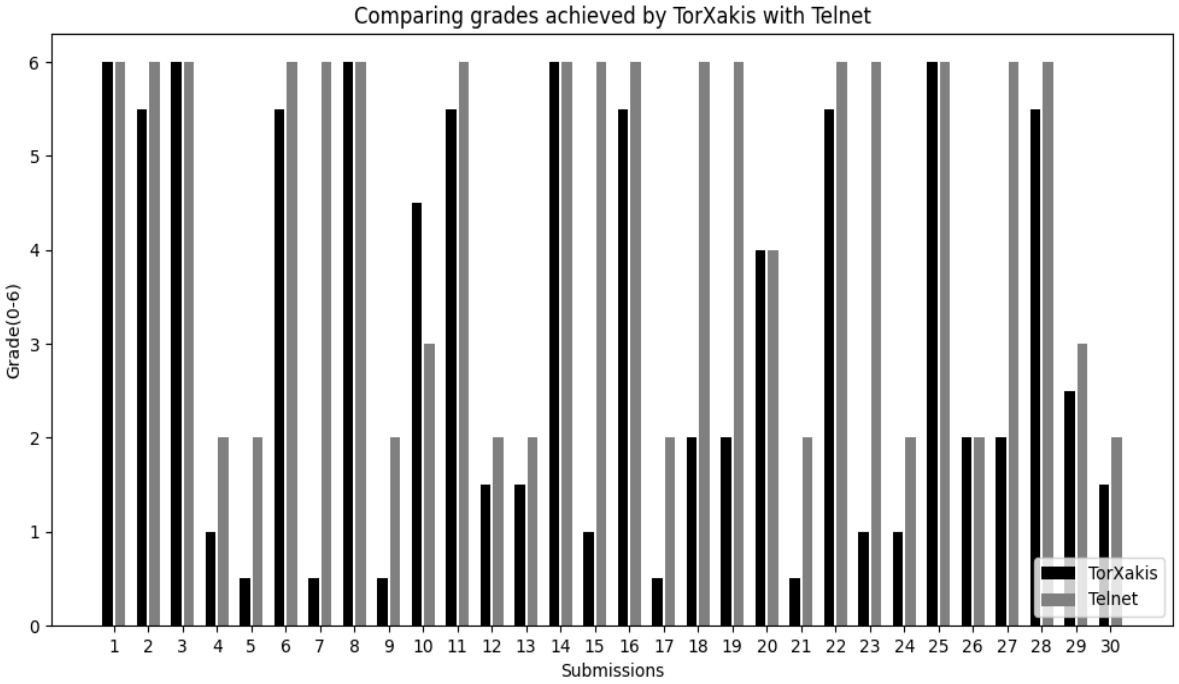

Figure 14. Comparison between TorXakis and Telnet

Figure 14, also indicates that despite the individual grade differences between two testing methods, most of the submissions received the same final PASS or FAIL results using both MBT and conventional methods.

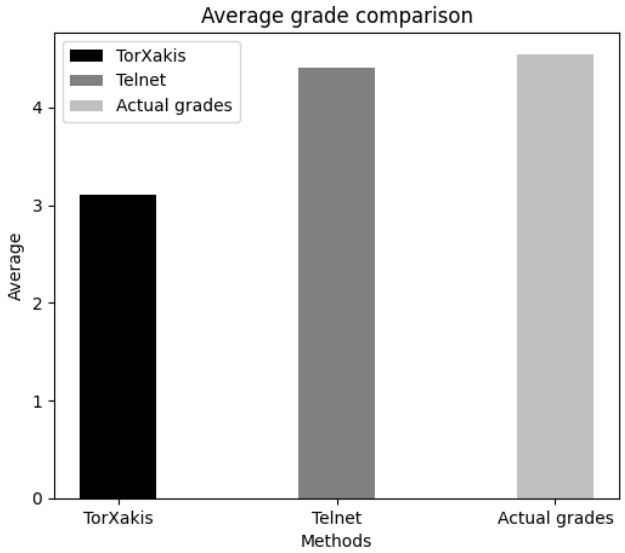

Figure 15. Comparing average of grades achieved

In Figure 15, we can see that TorXakis has a harsher approach compared to Telnet and actual grades while the other two are almost identical. Considering the similarity of pass and fail results, we can confidently say that despite how

harsher TorXakis is, it is still a solid option when it comes to time and money efficiency. While being harsher, this kind of grading is still fair and so certainly feasible.

This being said, we also note that the project considered here also included certain "non-functional" requirements. These kinds of requirements cannot be tested using model-based testing. One such a requirement was that concurrency management to the bulletin board file follow the readers-writers paradigm. A debugging mechanism involving read and write delays was required and then used in grading; these kinds of delays cannot be tested in TorXakis. Another example of non-functional requirement is the absence of busy waiting loops, which was marked by observing the system load during testing and raising a flag once the load exceeds a given threshold. Such a verification can be (and has been) easily done in an automated fashion with a simple shell script, but is certainly outside the realm of model-based testing.

An unexpected failure in TorXakis grading was testing synchronization. Most submissions failed with flying colors while being tested with TorXakis, yet some did receive substantial partial marks during human grading. Students were asked to expose the synchronization protocol when using a suitable debug switch and so the human instructor was able to observe the protocol in action during testing. Partial marks were given for attempts at communicating with various degrees of success despite the process eventually failing. It is conceivable that a suitable TorXakis model can act as a peer in the replicated system and thus observe the same application protocol, but the way the project considered here was formulated effectively prevents such an approach. Indeed, the design of the application protocol for synchronization was the task of the students, and so was different from submission to submission, thus preventing any kind of automated testing other than the one based on the end result. Whether a grade based solely on said end result is fair is obviously open to interpretation.

Finally, it is worth noting that the source code of submissions was inspected cursorily. This was done mostly to establish code ownership but at times was used to clarify strange behavior and possibly give partial marks for otherwise non-functional aspects of a submission. This is obviously well above the TorXakis' pay grade.

This all being said, as a side effect of this study we note that it is apparently unnecessary to test non-functional requirements separately. Indeed, we note in Figure 15 that the grade average of telnet testing (which

only tests functional aspects) is virtually the same as the actual grade (which includes all the non-functional criteria mentioned above). It would thus appear that the quality of the code and the correct execution of the code are strongly correlated. Obviously though this needs more investigation.

Bibliography

[1] Dewey, Patrick R. "*The Essential Guide to Bulletin Board Systems*" The University of Michigan, Meckler, 1987

[2] Tretmans. Jan & Van de Laar. Piërre. "*Model-Based Testing with TorXakis: The Mysteries of Dropbox Revisited*." In: 30th CECIIS, October 2-4, 2019, Varaždin, Croatia

[3] Tretmans. Jan. "*Model Based Testing with Labeled Transition Systems*." In: Hierons, R.M., Bowen, J.P., Harman, M. (eds) Formal Methods and Testing. Lecture Notes in Computer Science, vol 4949, 2008, Springer, Berlin, Heidelberg.

[4] Tretmans. Jan. "*Conformance Testing with Labeled Transition Systems: Implementation Relations and Test Generation."* In: Computer Networks and ISDN Systems, Volume 29, Issue 1, Pages 49-79, 1996

[5] Gaudel. Marie-Claude. *"Problems and Methods for Testing Infinite State Machines: Extended Abstract" ,* In: Electronic Notes in Theoretical Computer Science, Volume 95, Pages 53-62, 2004

[6] Driscoll. K. "*Social media's dial-up roots"* In: IEEE Spectrum 53 (11), 54-60, 2016

[7] Tretmans. Jan. "*A Formal Approach to Conformance Testing*" PhD thesis, University of Twente, Enschede, The Netherlands, 1992

[8] Wezeman.C. D. "*The CO-OP method for compositional derivation of conformance testers*" In: E. Brinksma, G. Scollo, and C. A. Vissers, editors, Protocol Specification, Testing, and Verification IX, pages 145–158, 1990, North-Holland.

[9] R. Khare "*TELNET: the mother of all (application) protocols*." IEEE Internet Computing, Volume: 2, Issue: 3, 1998

[10] Lina Khalid. "*Software Architecture for Business.*" Springer Cham Nature Switzerland AG, 2019 (pages 95-106)

[11] Yishuai Li, Benjamin Pierce and Steve Arthur Zdancewic. "*Model-based testing of networked applications.*" In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021). Association for Computing Machinery, New York, NY, USA, 529–539.

[12] Vahid Garous, Alper Buğra Keleş, Yunus Balaman, Zeynep Özdemir Güler, Andrea Arcuri "*Model-based testing in practice: An experience report from the web applications domain*" Journal of Systems and Software, Volume 180, 2021

[13] Weiglhofer, M., Aichernig, B.K. "*Unifying Input Output Conformance.*" In: Butterfield, A. (ed) Unifying Theories of Programming. UTP 2008. Lecture Notes in Computer Science, vol 5713, 2010, Springer, Berlin, Heidelberg.

[14] Artho, C.V, Armin, B, Masami H, Eric P, Martina, S, Yoshinori, T & Mitsuharu, Y. "*Modbat: A Model-Based API Tester for Event-Driven Systems.*" In: Bertacco, V., Legay, A. (eds) Hardware and Software: Verification and Testing. HVC 2013. Lecture Notes in Computer Science, vol 8244, 2013, Springer

[15] Brogi, A., Soldani, J., Wang, P. "*TOSCA in a Nutshell: Promises and Perspectives.*" In: Lecture Notes in Computer Science, vol 8745, 2014, Springer, Berlin, Heidelberg

[16] Holzmann, J. G. "*The Spin Model Checker: Primer and Reference Manual.*" Addison-Wesley, 2004

[17] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri. "*NuSMV: a new Symbolic Model Verifier*" In The International Journal on Software Tools for Technology Transfer (STTT) 2, 410–425, 2000

[18] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe. "*A Theory of Communicating Sequential Processes.*" J. ACM 31, 3 (July 1984), 560–599

[19] Milner, R. "*A Calculus of Communicating Systems.*" Lecture Notes in Computer Science, vol 92, 1980, Springer, Berlin, Heidelberg

[20] Joachim Parrow, "*An Introduction to the π-Calculus*" Handbook of Process Algebra, Elsevier Science, 2001, Pages 479-543