Applying Graphical User Interface guidelines to the Model Based Testing tool TorXakis

Author: Tobias Schröter  
s4122283

Supervisor: Dr. ir. G.J. Tretmans  
tretmans@cs.ru.nl

Second assessor: Peter Achten  
P.Achten@cs.ru.nl

March 25, 2017
Abstract

The Model Based Testing process of TorXakis can be improved by creating a Graphical User Interface for the tool. The use of a Graphical User Interface allows the user to create a model in a graphical way. In the old process of TorXakis the users often create a graphical model and afterwards cast it to the TorXakis-input language.

In software development processes the testing part is increasingly getting more attention. Model Based Testing offers a way to test systems automatically by using the specification of the system. Such systems are called SUT (System Under Test).

In this research a Graphical User Interface, GTorXakis, for TorXakis is created. Therefore design guidelines for the creation of such Graphical User Interfaces are analysed and design requirements for GTorXakis are created. In an experiment with users of TorXakis the usability of GTorXakis is analysed. The test persons were asked to describe the Model Based Testing process in their own words, so that the Model Based Testing process that is used as basis in this research can be validated.

The results of the experiment show that GTorXakis needs to be improved in some situations, but already offers a good usability. All in all GTorXakis solves the problem of creating two models by creating a graphical model that is automatically casted to the TorXakis-input language.
Preface

In 2015 I took part in the course Testing Techniques at the Radboud University in Nijmegen. In the course we have applied different techniques of testing, for example Model Based Testing. While applying it, we were asked to choose a Model Based Testing tool. My group members and me chose TorXakis.

In the process of using TorXakis I thought that the process can be improved by adding a Graphical User Interface, which automatically creates the models in the TorXakis-input language. I already had experience of developing such Graphical User Interfaces and I was interested how people of TorXakis will think about it.

GTorXakis depends on TorXakis. In the development process of GTorXakis, the type definitions of TorXakis changed, such that GTorXakis also needs to be updated. Fortunately, the developers of TorXakis added some important functionality for me to communicate with the tool over sockets. The experiment with the test persons was an interesting and funny experience for me. I am grateful to all the test persons, because they made it possible to run the experiment in a good and easy way. I would particularly like to thank my supervisor Jan Tretmans and second assessor Peter Achten, who supported me in an excellent way during the creation of this report. Furthermore I want to thank my family, friends and girlfriend for their support, especially in the end of the writing process.
# Contents

1 Introduction ................................................. 5  
  1.1 Current situation ........................................ 6  
  1.2 Solution approach ........................................ 7  
  1.3 Research question ........................................ 8  
  1.4 Outline of experiment .................................... 9  
  1.5 Motivation ............................................. 9  
  1.6 Roadmap ............................................... 9  

2 Theoretical background .................................... 11  
  2.1 Model Based Testing ..................................... 11  
    2.1.1 IOCO Implementation Relation ...................... 13  
  2.2 What is TorXakis? ....................................... 16  
    2.2.1 Architecture ....................................... 16  
    2.2.2 Syntax ........................................... 17  
    2.2.3 Testing methods .................................. 20  
  2.3 Model Based Testing Process ......................... 21  
    2.3.1 As part of development process .................. 21  
    2.3.2 As part of external testing process ............ 22  
  2.4 Usability test ......................................... 23  
  2.5 Design guidelines ...................................... 25
3 Related work
  3.1 Model Based Testing process .................................. 27
  3.2 Existing modelling tools ........................................ 28

4 GTorXakis
  4.1 Design ............................................................... 31
    4.1.1 Functional requirements .................................. 32
  4.2 Global information ............................................... 32
  4.3 Structure .......................................................... 33
  4.4 Run tests .......................................................... 36
  4.5 Other features .................................................... 38

5 Research method .................................................... 39
  5.1 Theoretical variables ............................................ 39
  5.2 Usability test ..................................................... 40
  5.3 Experiment ........................................................ 42
    5.3.1 Preparation ................................................... 42

6 Experiment results ................................................ 44
  6.1 Test persons ....................................................... 44
  6.2 Model Based Testing process .................................... 44
  6.3 Usability .......................................................... 47
    6.3.1 Questionnaire ................................................ 47
    6.3.2 Observation .................................................... 49
  6.4 Validity threats .................................................. 53

7 Conclusion and discussion ......................................... 54
  7.1 Conclusion ........................................................ 54
  7.2 Discussion ........................................................ 60

8 Appendix .............................................................. 63
Introduction

The Model Based Testing tool *TorXakis* is developed at the Radboud University Nijmegen and TNO-ESI\(^1\). Testing is gaining more and more attention during software development. For example Zhang et al. claim that in real-time embedded software, software testing "is gaining increasingly more attention" [13].

As a result Model Based Testing also gets more attention because it offers a way to automatically test a system with less time and a better coverage than manual testing. Model Based Testing is a way to test a given system with help of the specification. This technique does not only improve the code, but also the specification itself. For example it might be that a Model Based Testing Tool found a vulnerability that is a false positive, i.e. the system is correct but the specification has a fault. It is important to enhance the process of this testing technique, because the models can be easily modified if the program changes, it improves the code and specification and it finally tests the program by generating random paths through a model.

In my opinion *TorXakis* itself is a very impressive tool, but at the moment not really user-friendly. Users of *TorXakis* need a deep knowledge about the Model Based Testing process, because of the complexity of the tool.

The purpose of this research is to enhance the modelling process by developing a Graphical User Interface, called GTorXakis, where the user can easily create models and test a system. GTorXakis will be developed by applying

\(^1\)http://www.esi.nl
design guidelines to it. In an experiment the usability of GTorXakis will be measured. In addition to that the Model Based Testing process itself will be analysed by studying literature. The result of this literature research will be validated during the experiment.

1.1 Current situation

The process of Model Based Testing starts with reading the specification of the System Under Test (SUT). This is the starting point, because it is impossible to specify a model of a SUT without knowledge about that SUT. After that a sketch of the model is created by drawing it on paper. The next step is the validation of this sketch. Therefore the sketch will be compared to the specification. The comparison is done by the developer of that sketch.

TorXakis needs a model in the TorXakis-input language as input, thus the sketch needs to be transformed into this language. Because it is impossible to validate that the resulting model in the TorXakis-input language is correct, the model is again validated by the developer. Since the sketch is already validated, one can simply compare the new model with the sketch. If the model in the TorXakis-input language is also valid, the test-cases can

![Diagram]

Figure 1.1: Old process of Model Based Testing
be generated and executed. The testing process can find errors that are not errors in the SUT but in the model. In this situation, the error might be in the specification itself, in the sketch or in the TorXakis model.

The modelling process can be shortened if the user is very familiar with TorXakis and does not need a sketch to create the model in the TorXakis-input language.

An example of the current situation is shown in figure 1.1. Within the course Testing Techniques² at the Radboud University the students follow this process. In the next section a possible improvement of this situation is presented.

1.2 Solution approach

In the current situation of Model Based Testing with TorXakis two models were created (see fig.1.1). These two models need to be validated. A validation is very vulnerable to faults, because it can always happen that the person who validates the model does not find all the faults in the model. In the experience of the researcher it is not easy to create a model in the TorXakis-input language.

![Figure 1.2: New process of model based testing](http://www.studiegids.science.ru.nl/2015/course/37015/)

The idea of the adaptation is that the users create a graphical model within TorXakis.
GTorXakis. This program transforms a graphical model representation to the TorXakis-input language. Therefore a Graphical User Interface with the opportunity to draw a model will be developed. In addition to that, GTorXakis offers a way to run the developed model with TorXakis. Instead of learning the input language of TorXakis, users now need to learn GTorXakis.

1.3 Research question

The goal of this research is to build a Graphical User Interface for the Model Based Testing Tool TorXakis to improve the Model Based Testing process and make it more usable for users. So:

How can the Model Based Testing process of TorXakis be improved by designing a Graphical User Interface for it?

The research question will be answered by the following sub-questions:

1. What does the Model Based Testing process look like?

2. How does TorXakis support the Model Based Testing process?

3. Which parts of the Model Based Testing process need to be implemented in the Graphical User Interface?

4. Which Graphical User Interface design guidelines can be implemented?

5. Is the new Graphical User Interface usable for users of TorXakis?

Sub-question five will be answered by the results of an experiment that will be part of this research. Furthermore the result of the first sub-question is validated by the test persons from that experiment. The improvement of the Model Based Testing process will not be measured with a comparison between TorXakis and GTorXakis, but will be measured by analysing the Model Based Testing process and its sub-processes that can be improved by using a Graphical User Interface.
1.4 Outline of experiment

In the experiment of this research TorXakis users will use GTorXakis to create graphical models and test systems. The usability of GTorXakis is measured during the experiment, because the improvement of the Model Based Testing process depends on it. Furthermore the test persons will be asked to explain the Model Based Testing process, such that the Model Based Testing process, analysed in section 2.3, can be validated/adjusted.

1.5 Motivation

The idea of improving the Model Based Testing process is established within the course Testing Techniques where I had the possibility to test SUTs with TorXakis. I was not really familiar with the tool, therefore, I created the first model in a graphical representation (e.g. on paper) and transformed this model to the TorXakis-input language.

The strategy of the students was the motivation and kickoff to develop GTorXakis that transforms a graphical model to the TorXakis input language and improve the Model Based Testing process by removing redundant work, i.e. creating and validating two models.

Because testing is gaining more and more attention, the second motivation is to make the Model Based Testing process more clear for new users.

1.6 Roadmap

In the next chapter the theoretical background of Model Based Testing is explained. With this background information about Model Based Testing, TorXakis and the TorXakis -input language will be summarized. The analysis of the Model Based Testing process is also part of the chapter.

The third chapter will summarize related work on Graphical User Interface
design, Model Based Testing and usability testing. 
In chapter four design guidelines of Graphical User Interfaces will be outlined. Additionally the requirements, structure and functionality of GTorXakis are stated.
Beside the explanation of the research method, the structure of the experiment will be explained in chapter five. After that the results are summarized in the sixth chapter. In the last chapter the answers to the research question and all the subquestions are given.
Theoretical background

In this chapter the theory of Model Based Testing is explained and an introduction to TorXakis, the Model Based Testing process, usability testing and design guidelines for creating Graphical User Interfaces is given.

2.1 Model Based Testing

Model Based Testing (MBT) needs a System Under Test (SUT) and the specification of that SUT as input. As the name suggests, the testing technique is based on a model. This model is created out of the specification of the SUT. The resulting model describes the expected behaviour of that SUT.

The advantage of MBT compared with some other testing techniques is, that it not only provides automation of test execution but also the automation of generating the test cases. These test cases are generated from the model. An example of such an alternative testing technique is unit testing. It is often used in software development. The test cases need to be written by a human, often the programmer itself.

If the model describes precisely the behaviour of the SUT, then all test cases, that are crafted from this model, will be valid [11].

As result of automation, testing needs less time. Obviously the creation of the model needs a lot of time, but the creation and execution of a test case needs less time [8].

Model Based Testing is gaining more and more attention from the industrial
and academic perspective. From the industrial perspective it offers a way to reduce costs for testing and from the academic perspective it is an extension of formal methods and verification methods [11].

In general there are several types of testing. These types can be divided into three dimensions:

- Characteristics
- Level of detail
- Accessibility

In figure 2.1, taken from the course testing techniques, these dimensions are shown. Model Based Testing can handle different kinds of testing. Often it is used for black-box testing on system level that tests the functionality of the system [11]. In figure 2.2 this setup is shown. The whole SUT is the black box. This black box gets an input and generates an output. This output is checked against the expected output. If this output does not match, the test
has found an error.
In the context of testing there are different algorithms to walk through a model and test if an implementation is conforming to the specification. The algorithms will be called relations. The base of many of these relations is the IOCO Implementation Relation.

### 2.1.1 IOCO Implementation Relation

The Input-Output Conformance (IOCO) Implementation Relation is based on the fact, that an implementation is only ioco-conform, "if any experiment that is derived from the specification leads to an output from the implementation that is foreseen by the specification" [11].

![Figure 2.3: Example: IOCO Implementation Relation](image)

In figure 2.3, taken from the course testing techniques, two examples are shown. The example will use Labelled-Transition-Systems (LTS). In an LTS a transition with a question mark (e.g. \(?a\)) corresponds to an input, while a
transition with an exclamation mark (e.g. !x) corresponds to an output. In the context of a coffee machine, the input might be for example 1-euro and the output the coffee itself.

To explain the IOCO Implementation Relation, we will take as specification $s_1$ and as implementation $i_1$. The second step is to generate for every trace in $s_1$ the belonging set of outputs. The symbol $\delta$ defines quiescence, which means that there is no observable output. If an implementation has the possibility to be quiescent, then the specification needs to have this possibility too [11]. Moreover the symbol $\epsilon$ defines the empty trace. The definition of the functions $out$ and $after$ can be find in [11].

\[
\begin{align*}
out(s_1 \ after \ \epsilon) &= \{\delta\} \\
out(s_1 \ after \ \delta) &= \{\delta\} \\
out(s_1 \ after \ \?a) &= \{x, y\} \\
out(s_1 \ after \ \?a!x) &= \{\delta\} \\
out(s_1 \ after \ \?a!y) &= \{\delta\}
\end{align*}
\]

The next step is to create the same sets for the implementation. Therefore we will check the traces of the specification, that means traces which are in the implementation but not in the specification are not considered:

\[
\begin{align*}
out(i_1 \ after \ \epsilon) &= \{\delta\} \\
out(i_1 \ after \ \delta) &= \{\delta\} \\
out(i_1 \ after \ \?a) &= \{x, y\} \\
out(i_1 \ after \ \?a!x) &= \{\delta\} \\
out(i_1 \ after \ \?a!y) &= \{\delta\}
\end{align*}
\]
Because all sets of \( i_1 \) are subsets of the sets from \( s_1 \), the implementation \( i_1 \ioco s_1 \). However it is important that we only check the traces of the specification, not of the implementation.

**Definition**  The precise definition of the IOCO Implementation Relation is:

\[
i \ioco s \iff \forall \sigma \ Straces(\sigma) : \text{out}(i \after \sigma) \subseteq \text{out}(s \after \sigma) \quad (2.1)
\]

The set of all traces of the specification is called in the definition \( Straces(s) \) and the function \( \text{out} \) calculates a set of all output values after a trace \( \sigma \).

If we look at the implementation \( i_2 \) and the specification \( s_1 \) we can show that \( i_2 \ioco s_1 \), because \( \text{out}(i_2 \after ?a) = \{x, z\} \) is not a subset of \( \text{out}(s_1 \after ?a) = \{x, y\} \).

**Other Implementation Relations**

There are many other Implementation Relations and some of them are based on the IOCO Implementation Relation. For example:

- \( uioco \)
- \( ioconf \)
- \( TGV-ioco \)
- \( mioco \)

In [11] these and other relations are presented and explained. This research will not focus on these relations, because TorXakis uses the IOCO Implementation Relation.
2.2 What is TorXakis?

TorXakis is developed at the Radboud University Nijmegen and TNO-ESI\footnote{http://www.esi.nl} and new releases are published on a regular basis. The tool offers a way to execute and generate test cases automatically. Moreover TorXakis offers different testing methods that will be explained at the end of this section. The TorXakis-input language is mainly based on the ISO specification language Lotos \cite{2}. Lotos is a specification language that is developed during the years 1981-1986. The language offers a way to describe how processes can be modelled in a system.

TorXakis offers the opportunity of ”on the fly” testing. ”On-the-fly” testing means that the tool generates the test cases while executing the test. In Model Based Testing, some tools only offer the generation of the tests and execution in two different steps.

In an old version of TorXakis (Ver. R269), the syntax of the TorXakis-input language consists of much redundancy, which makes it more complex to model with it. In section 2.2.3 this redundancy is explained. During the process of developing GTorXakis, a new version of TorXakis was released (Ver. R803). In the new version of TorXakis the developers added a new definition to the TorXakis-input language, the STAUTDEF.

2.2.1 Architecture

The structure of TorXakis is shown in figure 2.4. TorXakis derives the test from the model definition and communicates with the SUT. This communication is defined by a connection definition that maps the TorXakis representation to the SUT representation and vice versa. The communication between the tests and the communication definition happens over two channels as well.
2.2.2 Syntax

In *TorXakis* the user can define seven different types of definitions. Two of these definitions can be used to define a model:

- **PROCDEF** → Process definition
- **STAUTDEF** → State automaton definition

In a PROCDEF the user defines the model as a set of processes and in the STAUTDEF the user defines the model as a state machine.

![Image of TorXakis diagram](image)

Figure 2.4: Simplified structure of *TorXakis*

### Example

```plaintext
STAUTDEF example [ Channel1 :: Int; Channel2 :: Int # Int; Channel3 :: Int ]

::= 
  STATE state1, state2
  VAR variable1 :: Int; variable2 :: String
  INIT state1 { variable1 := 0, variable2 := ”Hello world” }
  TRANS state1 → [variable2 = ”Hello world”] Channel1 ?x 
  { variable1 := x } → state2 
  state2 → Channel2 ?x ?y 
  { variable1 := x - y } → state2 

ENDDEF
```

Figure 2.5: STAUTDEF with two states and three variables

In the STAUTDEF type definition, the user can create a list of states, a list of variables and a list of all transitions in that model. Every transition can
have a condition and an action that is triggered if the transition is executed. An example of this definition is shown in figure 2.5. The list of all states need to be defined after the STATE identifier. In this example there are two states: state1 and state2. Equal to the STATE identifier there is an identifier for all variables in that system.

INIT identifier that is used to initialize the variables and define the start state, the transitions are defined after the TRANS identifier. In this example there is a transition from the state state1 with the condition that variable2 equals "Hello world". Furthermore there needs to be an input to channel1. In the "{}" brackets actions are defined. In this example x will be assigned to variable1. The graphical representation of the STAUTDEF in figure 2.5 is shown in figure 2.6.

![Graphical representation of the STAUTDEF from figure 2.5](image)

Figure 2.6: Graphical representation of the STAUTDEF from figure 2.5

If the user wants to define the model with PROCDEFs, every state in the model is represented by a process. Modelling the state machine shown in figure 2.6 will need two PROCDEFs. An example of such a PROCDEF is shown in figure 2.7. In this example a process with the name tmp.1 is created. In section 2.2 we introduced that the TorXakis-input language has a redundancy in the syntax. A list of all possible channels that can be used need to be forwarded to every PROCDEF. If a process will call another process, the whole list need to be added too.
Figure 2.7: Process definition

In figure 2.4 the models are the input for the model definition. This model definition defines how the model that is used in the test behaves and which channels are used. In figure 2.8 the model definition of an Adder is shown. In a model definition channels can be specified as output and input. In the example the channel Plus, Minus are input channels. The channel Result is an output channel. The behaviour of a model is described after the BEHAVIOUR identifier. In this example the model will simply start the adder process. The process can be defined by a PROCDEF or a STAUTDEF.

Figure 2.8: Model definition

The communication between the SUT and the model is specified with a connection definition. In TorXakis this is called CNECTDEF. It maps the output and input between these two worlds. In figure 2.9 this definition is shown. In the example two channels are declared. One channel is used for the output of TorXakis and one channel is used for the input. Every channel needs to know its type and which host and port should be used. Furthermore the encoding and decoding of the communication is defined.

With these definitions TorXakis is able to test a SUT with a model. Optional definitions that might be very helpful are FUNCDEF, TYPEDEF and CONSTDEF. A FUNCDEF is used to define a function. Examples are the
functions `toString` and `fromString` defined in figure 2.9. Besides the TYPE-DEF that is used to define own datatypes, for example the type `Operation` used in figure 2.9, a CONSTDEF is used to define constants that can be used in every definition.

### 2.2.3 Testing methods

*TorXakis* offers different ways of testing System Under Tests. The tool has the possibility to step through the defined model without connecting to the real SUT. This method can be very helpful to check if the model contains errors, for example deadlocks. Another advantage of that approach is, that the SUT is not needed for testing. This testing method is called STEPPER. The second testing method of *TorXakis*, called TESTER, connects to the SUT and steps through the model and waits for the result of the SUT. In this situation the expected result is compared to the result of the SUT. The advantage of that testing method is that it tests the SUT and not the model that was built with *TorXakis*.

*TorXakis* offers the possibility to simulate the SUT. In this situation one instance of *TorXakis* is started in TESTER mode and one instance in SIMULATOR mode. The benefit of this testing method is, that it offers the opportunity to test mutations of the model and check if that is also conform to the specification of that model. In the coffee machine example from section 2.1.1 this could be for example the addition of an espresso.
2.3 Model Based Testing Process

The Model Based Testing Process is a concatenation of three sub processes. First of all there is the process to build a valid model (process described in introduction), to build the SUT and the process of testing with the model and SUT.

The Model Based Testing process that we want to define, is based on modelling with TorXakis. TorXakis needs the MODELDEF, the CNECTDEF and at least one PROCDEF or STAUTDEF as input, that represent the model of the SUT. The goal of Model Based Testing with TorXakis is to check that this model and the SUT are ioco conform (see 2.1.1). The model will be created with the specification of the SUT.

The creation of such a model is highly vulnerable, because the specification needs to be unambiguous. If the specification contains obscurities, the model and SUT might be implemented in different ways. During the testing process errors will be found, that are not in the SUT, but in the model.

2.3.1 As part of development process

If the Model Based Testing process is part of a development process the implementation process of the SUT is also part of the process. If the test fails and an error is found in the SUT, this error can be fixed by the developers.

In figure 2.10 a resulting Model Based Testing process is shown. This process is created with the information from section 3.1 and the process that was introduced in section 1.1. The process in figure 2.10 assumes that the Model Based Testing process is part of a development process. Every element in the process that has the shape of an octagon is a result and every circle an action.

The Model Based Testing process requires at the beginning the specification of the SUT. From this specification the SUT and model will be created. If the model is deemed valid, the test generation and execution will start. The
validation of a model is done by comparing the model and the specification. If the results contain errors, the SUT will be changed and the test will be executed again. If the results do not contain errors, the testing process is over.

Obviously it is also possible that the model is not valid, therefore the edge between Error found and Specification is made, because it might be, that the specification contains also errors. As a result, the model and/or SUT needs to be re-validated.

![Model Based Testing process as part of a development process](image)

Figure 2.10: Model Based Testing process as part of a development process

### 2.3.2 As part of external testing process

The Model Based Testing Process shown in figure 2.10 is one possible process. For example if an external test company should test a system, the process changes. The company gets the SUT as black-box and the specification which is used by the developers. The test company will create a report of the errors, that are found during testing. In this variation the SUT will be changed if the developers get the report of the test company. Furthermore it is possible
that the errors are not in the SUT, but in the model or specification. It might be that the specification is not up-to-date, thus the SUT is made with another specification as the model. In figure 2.11 this variation is shown. This process is also used at the Radboud University in the course Testing Techniques.

Figure 2.11: Model Based Testing process as part of external testing process

### 2.4 Usability test

The usability of a system can be measured by different usability scales. In [1] different usability scales are compared. The System Usability Scale (SUS) was created by [3] and adapted by Aaron Bangor et al. in 2008. It is a questionnaire that is used to measure the usability of a system. In [1] they analysed the old questions from [3] and changed some words so that the questions will be better understood, e.g. the word *cumbersome*. *Cumbersome* is an older word that is changed to *awkward* because it is used more often (Oxford University Computing Service, 2001).
In addition to that [1] compares different usability scales by their reliabilities. The System Usability Scale provides an algorithm to calculate the usability based on the answers from the questionnaire. The algorithm is described in [3]. The result of the algorithm is a numeric value between 0 and 100, where 100 corresponds to a perfect usability and 0 to totally unusable.

Alternative usability scales that are also stated by [1] with a better reliability need to be filled in online. Two alternatives that can be answered on paper too are [6] and [5], where [6] has an unreported reliability and [5] consists only of three questions.

Another strategy to measure if a system is usable are the Human Factors that were created by [10]. The author created five different measurable factors:

- Time to learn
- Speed of performance
- Rate of errors by users
- Subjective satisfaction
- Retention over time

The first human factor, *Time to learn*, is typically measured with the time that the test persons need to do some actions for the first time. The second human factor, *Speed of performance* is measured with the time that the test persons need for a given task. *Rate of errors by users* are measured with the errors that will occur during the execution of some tasks. The *Subjective satisfaction* need to be measured with an interview or questionnaire. The last human factor is *Retention over time*. This human factor is used to measure if the test persons will maintain their knowledge after a given time, e.g. a month.
2.5 Design guidelines

In [4] user interface design is discussed. The author says that ”The success of any computer application is depending on it providing appropriate facilities for the task at hand in a manner that enables users to exploit them effectively.” [4]. Furthermore does Dillon et al. say that ”It is not yet possible to talk of a complete theory of human-computer interaction,[...]” [4].

Dillon says, that the program always communicates to the user by appropriate feedback. For example if the user adds a state, the program needs to show this state. In addition to that, the author claims that a perfect feedback takes the user ”closer to their goal as a result” [4]. An example that shows ”perfect” feedback could be, that the program shows the result of the testing step in different colors, i.e. a positive result is shown with green and a negative result is shown with red. Furthermore the feedback needs to be clear and informative.

In the learning process of a new system, the feedback is important and also that the user can undo actions. For example if he/she triggers an action that was not planned [4].

Design guidance recommends to show dark on light, i.e. the background should be light (e.g. white) and everything on that background should be

<table>
<thead>
<tr>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always appropriate feedback</td>
</tr>
<tr>
<td>Undo-/Redoing from actions</td>
</tr>
<tr>
<td>Use of icons</td>
</tr>
<tr>
<td>Background color lighter</td>
</tr>
<tr>
<td>Foreground color darker</td>
</tr>
<tr>
<td>Consistent</td>
</tr>
<tr>
<td>Knowledge about TorXakis should not be redundant</td>
</tr>
</tbody>
</table>

Table 2.1: Design requirements
darker (e.g. black, grey) [4]. In addition to that, Dillon says it is good to use images and symbols for actions, e.g. the floppy disk icon that is used in every application as symbol for saving documents. An other example is the folder icon that is used for opening projects. The use of icons enables international use, because the icons are language independent.

Another guideline of user interfaces design is that the system has a consistent steering, i.e. same kind of actions will be triggered every by the same key [4]. For example if the user can change the name of a state by the key \textbf{F2}, the user can also use \textbf{F2} to change the name of an edge.

If a new version of an older system will be released, the knowledge from the previous users should not be redundant, i.e. they do not need to learn the new system from scratch [4]. In the case of \textit{TorXakis}, that means that the user should be able to use his/her knowledge about the old syntax.

The design requirements of G\textit{TorXakis}, which are based on these design guidelines, are shown in table 2.1.
Related work

In this chapter related work on Model Based Testing is explained. Furthermore common tools that enabled the creation of graphical models are analysed. The tools that are the base for this analysis are yEd\(^1\) and Uppaal\(^2\).

3.1 Model Based Testing process

The Model Based Testing process is built with the information gained from [12, 7]. In [12] automated testing in the context of Model Based Testing is discussed. The authors claim that: ”Specification-based refers to the existence of a specification which exactly prescribes what the system shall do and what not. This specification is the starting point for testing.” [12] Furthermore they claim that ”With formal, model based testing the specification is given as a model in some formal language. This formal specification is the starting point for testing the sut.” [12] Thus the starting point for Model Based Testing that is based on a specification, is the specification itself. The model that is used in the Model Based Testing process needs to be specified in a formal language. While using TorXakis this formal language is the TorXakis-input language.

Moreover [12] say that test generation and execution is part of automated

\(^1\)http://www.yworks.com/products/yed
\(^2\)http://www.uppaal.org/
testing. [7] created a modeling and testing process of another tool: Spec Explorer\(^3\). Within this process the test case execution and test generation are both part of the process, too. In the shown process it is assumed that the SUT is working correctly if there are no longer errors during testing.

### 3.2 Existing modelling tools

**yEd** is a graph editor where users can create different kind of models, whereas **Uppaal** is a modelling tool which can validate and test systems. In both

![Figure 3.1: Steering of yEd](https://msdn.microsoft.com/en-us/library/ee620411.aspx)

---

\(^3\)https://msdn.microsoft.com/en-us/library/ee620411.aspx
systems the user is able to create states and transitions between those states. Furthermore the user can select states or transitions to delete or move them. 

yEd is used as modelling tool for the Model Based Testing Tool GraphWalker\(^4\). In detail GraphWalker has no opportunity to create models. The models will be created in yEd (with a given syntax). For the testing process these models will be imported in GraphWalker.

![Diagram](image)

**Figure 3.2: Steering of Uppaal**

In figure 3.1 the steering of yEd is displayed as a simple state machine. This state machine is created by analysing the behaviour of yEd. The start state of the yEd-steering is that nothing is selected, in the figure this state is called None. From this point the user of yEd can add new elements, select elements and create transitions between elements in the model. If the user selects elements of the model (e.g. through a selection box), it switches to a new state. This state is called Selected. In this state the user is able to move elements or go back to the None state. The octagons in the figure are used

\(^4\)http://graphwalker.github.io/
as temporary states. Those states are triggered by an action from the mouse and are left by releasing the mouse.

The steering of Uppaal is shown in figure 3.2. In Uppaal the user selects in a toolbar which action he/she wants to take. The state in which the system is, can only be changed through this toolbar.
**G TORXAKIS**

GTorXakis is a standalone application that enables the graphical creation of models that can be used in the Model Based Testing tool *TorXakis*. A model is a set of states and edges. In [10] different application domains are discussed. Shneiderman [10] defines the domain for systems that are in the field of: software development and scientific modeling systems as exploratory, creative, cooperative system, because GTorXakis itself is a modelling system in the software development cycle.

The states in GTorXakis have the shape of a circle, like the states in *Uppaal* and *yEd*. A transition between two states is displayed with an edge. The application does not only allow to create graphical models, but rather allow to fully specify all type definitions and the communication with *TorXakis*. Via a socket communication GTorXakis connects with *TorXakis*. This structure is chosen, because it allows to communicate in the future with a *TorXakis* server.

### 4.1 Design

The design requirements of GTorXakis are explained in section 2.5. In this section functional requirements of GTorXakis will be stated.
4.1.1 Functional requirements

GTorXakis should enable the graphical creation of models and offer the opportunity to specify all the type definitions of TorXakis. Therefore GTorXakis will need a drawing area where these models can be created. The drawing area of GTorXakis will implement the steering system of yEd, because it is more dynamic than the steering system of Uppaal, i.e. the user does not need to choose the action he/she wants to trigger. The type definitions such as PROCDEF and MODELDEF (see section 2.2) will need text areas. These text areas should have syntax highlighting, which makes the developing process of the textual definitions easier.

Another functional requirement of GTorXakis is the communication with TorXakis. The tools should use sockets for the communication. GTorXakis should automatically communicate with TorXakis such that the user will only see the result of the communication.

TorXakis is developed platform independent and GTorXakis should be developed in the same way.

4.2 Global information

GTorXakis is developed in Java\textsuperscript{1}. Java is chosen because it is the preferred programming language of the researcher. Moreover it enables cross platform programming such that the program can run on every known operation system. This was an important requirement because TorXakis does that too. In section 4.1.1 the requirement of a drawing area is stated. The program provides a drawing area to draw the models. The requirement for the graphical creation of models is a steering system that allows this creation. The drawing area will implement the steering system of yEd shown in figure 3.1. Therefore we found several libraries that can be used to build such an area.

\textsuperscript{1}\url{https://java.com}
In table 4.1 three libraries that are free to use are shown.

<table>
<thead>
<tr>
<th></th>
<th>SVGSalamander²</th>
<th>Apache Batik³</th>
<th>JGraph⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good first impression</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Good documentation</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Easy to build &quot;Hello world&quot; program</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Made for creating models</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Experience</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 4.1: Assessment java libraries

SVGSalamander and Batik are SVG⁵ (Scalable Vector Graphics) libraries that are made for drawing SVG elements in a java program. With these libraries it is possible to build a SVG document and manipulate it while running the program.

JGraph is a library that is really made for drawing and creating models, but the first impression of the models that can be built with this library was not good. It was too difficult to understand which action can be made and how this action can be triggered. Although the Apache Batik library is not easy to use, it offers most functionalities which is the main reason for using this library. In addition to that the researcher has used the library already in a different project.

### 4.3 Structure

In GTorXakis the user has as main window a tabbed pane. In figure 4.1 this pane is shown. Every tab represents one type of definition, except the graphical models where every model is represented by a separate tab. For instance all FUNCDEFs are written in the tab *FUNC Definition*. In GTorXakis there are two different kinds of tabs: textual- and graphical ones. A textual tab

²https://svgsalamander.java.net
³https://xmlgraphics.apache.org/batik/
⁴https://github.com/jgraph/jgraphx
⁵https://www.w3.org/TR/2003/REC-SVG11-20030114/
contains a text area where the user simply enters the definitions as text. In
a graphical tab the user can draw a model.

The textual tab is made with the library called RSyntaxTextArea\(^6\). This
library adds the functionality of a text area with syntax highlighting and
makes it possible to add new syntaxes. In GTorXakis the syntax of \textit{TorXakis}
is added. In figure 4.2 the text area with some sample code is shown.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{rsyntaxtextarea.png}
\caption{GUI: RSyntaxTextArea}
\end{figure}

The graphical tab shows a drawing area to the user. As already mentioned
this drawing area is made with the library Batik. In figure 4.3 this drawing
area is shown. The steering of the drawing area is based on the modelling
tool yEd.

\(^6\)https://github.com/bobbylight/RSyntaxTextArea
In many modelling tools (e.g. yEd, Uppaal) the situation shown in figure 4.4 is allowed. In GTorXakis this situation is not allowed and is restricted to one edge. This edge is a representation of all edges, of the same direction, between these two nodes. The reason for that is that many edges will result in a confusing situation, because these edges might interleave. The managing of these edges is done in a dialog, where the user is allowed to add as many different transitions between the two states. In figure 4.5 this dialog is shown. The user needs to specify which gate is used and he/she can also specify a condition and action. In the "Action" field the user might want to
change a global variable or call a function that he/she defines in the "FUNC Definition" tab.

![Figure 4.5: GUI: Transitions dialog](image)

In TorXakis a STAUTDEF needs as input all gates. These gates can be specified in a sitebar in GTorXakis. This sitebar is shown in figure 4.6. A gate needs a unique name and a type as information. A STAUTDEF can handle global variables. These variables can also be added and edited in the sidebar. A variable needs a unique name, a type and an initial value.

### 4.4 Run tests

In section 2.3 the Model Based Testing process is discussed. In GTorXakis and TorXakis many steps of this process are implemented. In GTorXakis the user can run the test by opening the run dialog. The user has the opportunity to choose between three options:

- Stepping through the model without connection to the SUT
• Testing the SUT by communicating with it
• Simulating the SUT

![Figure 4.6: GUI: Sitebar of GTorXakis](image)

For the first two options the user can choose how many steps TorXakis will execute. If TorXakis finds an error, the user will be informed by a red error message. If the user is very familiar with TorXakis, he/she has the possibility to send other commands by using the text field at the bottom. In figure 4.7 this dialog is shown.

As already mentioned connects GTorXakis with TorXakis via a socket communication. The configurations of this communication of GTorXakis and TorXakis can be configured by the user. The possible communication between the SUT and TorXakis is configured in the CNECTDEF. If there are new versions of TorXakis the user can choose the directory of this new TorXakis server. Furthermore the user is able to choose the MODELDEF and CNECTDEF that should be used by TorXakis.
4.5 Other features

GTorXakis offers other features which are not necessary for testing but helpful for documenting. These features are:

- Exporting project to *.txs file
- Save project to *.gtxs
- Exporting graphical model to SVG, PNG, JPG

For example the testers can export the used model to JPG and include it in the test report. Moreover the users are able to save and open projects to work on those over more days. In addition to that GTorXakis offers the possibility to export the project to the fileformat *.txs that is used by TorXakis. If users want to export their project to the TorXakis-input language, they are able to do this. This feature is interesting for familiar users of TorXakis so that they can edit the models in the TorXakis-input language. Aside from that GTorXakis implements a history of all actions in the drawing area, that are trigged by the user. As a consequence the users are able to use an undo-and redo-function.
Research method

This research uses an experiment to analyse the Usability of GTorXakis. In this chapter the structure of this experiment and how the Usability will be measured within this experiment, are explained.

5.1 Theoretical variables

In a research project, the answer of the research question will be supported by theoretical variables. These variables should be indicated by at least one indicator. Indicators explain how theoretical variables can be described. These indicators are not directly measurable. Therefore every indicator will be judged by some measurements. In table 5.1 this structure is shown.

The theoretical variable "Usability" from table 5.1 is indicated by the System Usability Scale (SUS) [3], the Human Factors from [10] and the utility. The utility of the system will be tested by questions that were inspired by [9]. The SUS and the Human Factors are used to measure the usability of a system, where as the SUS uses a questionnaire and the Human Factors the behaviour of the test persons.

The utility of GTorXakis is measured by a comparison with TorXakis, because GTorXakis will be an addition to TorXakis. If GTorXakis is not valuable, it might be that TorXakis is more usable. With help of this theoretical variable it is possible to find an answer to the research question.
<table>
<thead>
<tr>
<th>Theoretical variable</th>
<th>Indicators</th>
<th>Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>System Usability Scale (SUS)</td>
<td>SUS result</td>
</tr>
<tr>
<td></td>
<td>Human Factors</td>
<td>Time to learn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speed of performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rate of errors by users</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subjective satisfaction</td>
</tr>
<tr>
<td>Utility</td>
<td></td>
<td>Answers on Q11-Q17</td>
</tr>
</tbody>
</table>

Table 5.1: Theoretical variable

### 5.2 Usability test

The first indicator in table 5.1 is the System Usability Scale that is introduced in section 2.4. This research will use the adapted System Usability Scale by [1], because the System Usability Scale is often used in usability studies following [1].

The second indicator of the theoretical variable *Usability* are the Human Factors that were presented in 2.4. The Human Factor *Retention over time* is not used, because the test persons will use the program only at one moment and it is not in the scope of this project. The Human Factor *Subjective satisfaction* will be measured by the result of the SUS. The other three Human Factors will be analysed with help of a screen and audio recording from every test person. The test persons will get a manual with detailed tasks for the experiment. Every time they start with a new task, the test persons will say that. With this information the speed of performance can simply be measured. In addition, the time to learn and the rate of errors can also be measured with help of the screen recordings. The utility of the system is the last indicator of the usability variable. The utility of GTorXakis is defined by the value that GTorXakis adds to *TorXakis* itself, i.e. if the new modelling process is better than the old one. The following questions are created with inspiration of [9], where the authors created questions to compare tools:

- It is easier to model with GTorXakis than it is with *TorXakis*
• The modelling process of GTorXakis is clearer
• GTorXakis does not help at all
• I would like to work with GTorXakis and not with TorXakis
• The Model Based Testing process is implemented in a good way
• It is easy to change something in the model
• It is easier to test a SUT with GTorXakis instead of TorXakis

The questions are directed to a comparison between GTorXakis and TorXakis, because the GUI is an adaption of TorXakis and should improve it. In table 6.3 the questionnaire of the research is shown. The original questionnaire is added to chapter 8 (Appendix).

Observation

As already mentioned the screen of the computer and audio will be recorded during the whole experiment to measure the three Human Factors: time to learn, speed of performance and rate of errors by users. For the Human Factor time to learn essential tasks of GTorXakis are observed. For example how long does it take for the test persons to learn how to add a state, draw an edge between states, rename the model and run the project with TorXakis. The test persons have no preliminary knowledge how GTorXakis works. As a consequence the time, how long it takes for the test persons to learn essential tasks, can be easily measured. The second Human Factor speed of performance is observed by analysing the time test persons need to perform given tasks from the manual. The observation of the second exercise in the experiment (see 5.3.1) is mainly used to measure how long test persons need to fully rebuild given models in GTorXakis. Furthermore the second exercise is used to determine which actions are implemented in a good way and which actions can be improved. An example of such an action might be
the renaming process of a model. The last Human Factor that is measured by an observation is the rate of errors by users. In this research we will not focus on the rate of errors by users, but on errors that occur during the experiment, e.g. the test person wanted to add a state and added an edge. The reason is, that it is difficult to measure the rate of errors, because every single action of the test person needs to be analysed. It is difficult to know if an action is wanted by the user or not. The only opportunity is to check if users "think loud". If there are test persons that are quieter, than this will not work correctly.

5.3 Experiment

The experiment is divided in two parts. In the first part the test persons will work with GTorXakis. This part should test if the program is usable for users with a higher knowledge about Model Based Testing or TorXakis. The test persons will be selected by their knowledge about TorXakis, because GTorXakis reuses the syntax and basics of it. Every test person will get 60 minutes to follow two exercises. In the first exercise the test persons build a simple model and in the second exercise a more difficult one. In the second part of the experiment, the test persons will be asked to answer the SUS questionnaire and the questions that are used to measure the utility. Furthermore the test persons are asked to explain the Model Based Testing process in their own words, such that the Model Based Testing process of this research can be validated.

5.3.1 Preparation

In the experiment the test persons will use the newly developed drawing area to build the models. In the whole experiment the test persons do not need to write the MODELDEF and CNECTDEF, so that they can fully concentrate
on the models. Another reason for this setup is that the test persons do not need to know how the System Under Test (SUT) communicates, but how the SUT works. If the test persons want to change these definitions, they are allowed to do it. In addition to that, the test persons get a manual with detailed tasks.

**Adder**  In the first exercise the test persons will create a model of a simple adder. This system is chosen because it is often used as simple example in Model Based Testing and it is possible that the test persons will solve it in less time. The MODELDEF and CNECTDEF which are used by the adder are shown in figure 2.8 and figure 2.9.

**Alternating Bit Protocol**  The second exercise is about the Alternating Bit Protocol (ABP). The test persons will get the definition of the ABP. The ABP is modelled with four sub-models. The sender, the receiver and two communication channels. The sender- (fig. 6.4) and one communication channel (fig. 6.5) model are given in the manual, but need to be added into GTorXakis. The test persons will get two of the four sub-models, such that the users know better what is expected by this exercise. Furthermore the test persons will use the simulator mode of TorXakis to simulate the SUT and test it.
Experiment results

In this chapter the results of the experiment are summarized. First of all the test persons of the experiment will be presented. After that the results of the experiment are constituted. The results of the experiment can be requested under specific circumstances, because of privacy reasons.

6.1 Test persons

The test persons of the experiment were experts of Model Based Testing and three of them were also experts of TorXakis. At this moment there are not many users of TorXakis. Two test persons had a slight knowledge about the tool and one test person had not any knowledge about it. Test person f had experience with yEd. In table 6.1 these test persons are presented. Test persons a and b completed only the first exercise, because they only had time for this one. The other test persons completed the two exercises.

6.2 Model Based Testing process

The test persons were asked at the end of the experiment to explain in their own words how the Model Based Testing process looks like. Four of the six test persons answered this question.
Test person a described the MBT-process with the state machine shown in
Table 6.1: Test persons

<table>
<thead>
<tr>
<th>Test person</th>
<th>Model Based Testing experience</th>
<th>TorXakis experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>c</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>d</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>e</td>
<td>X</td>
<td>(X)</td>
</tr>
<tr>
<td>f</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1. This state machine is similar to the first exercise description of the experiment and does only contain the modelling process of the Model Based Testing process.

The description of the MBT-process by test person b is as follows:

1. Find part or interface of SUT which is suited to MBT
2. Find specification, e.g. a document or discuss with developers
3. Find a tool and modelling paradigm that fits the specification/SUT
4. Modelling: iteratively model parts of the specification and try on the test that they generate

It describes the process of MBT and how it should be applied.

The third description of the model based testing process is from test person f. He describes the test for MBT as follows:
1. Define the states of the system

2. For each state define possible transitions on input stimuli

3. Associate each transition with SUT actions

4. Configure the test generation:
   - load statemachines
   - generate type of tests
   - define/configure MBT-tool

The MBT-process described by test person d is shown in figure 6.2. The test person characterized the MBT-process globally for TorXakis using the Stepper and Adder mode of TorXakis. In addition to test person a, test persons d and f do not only describe the modelling process but also the testing process. Test person b describes the Model Based Testing process in a more global context. In comparison with the developed Model Based Testing process shown in section 2.3, test person b does not specify the specification as a document, but also as a possible discussion with the developers. From
the discussion with the developers a specification of the SUT is created such that it ends up in a document too. Furthermore does test person b also include the retrieval of the part/interface of the SUT to the Model Based Testing process. The inclusion of the retrieval process might be a good variation, if the interface or part under test is not specified. Furthermore the test person adds the finding process of the tool to the Model Based Testing process. In the Model Based Testing process constituted in this research TorXakis is used as a tool, but it might be substituted with other tools.

6.3 Usability

In this section the results of the usability questionnaire and the observation are summarized.

6.3.1 Questionnaire

General As already mentioned the System Usability Scale contains ten questions. With the results of these questions and the algorithm provided by [3] it is possible to compute a value that represents the usability of the system. In table 6.2 the result of the SUS by every test person is represented. Both the average SUS score and the median are 73,75. In table 6.3 the average and median result of the questions are shown, whereas questions Q1-Q10 are taken from the System Usability Score questionnaire.

The test persons were able to answer the questions on a scale between 1 and 5, where 5 corresponds to strongly agree and 1 to strongly disagree. The average result to the question (Q3), whether the system is easy to use, is 3,67 and the median is 4. The test persons answered question Q7 on average with 4,17 and a median of 4, which corresponds to the answer agree. Question Q2 tests if GTorXakis is complex. The test persons answered this question on average with disagree. Question Q6 tests if GTorXakis has too much inconsistency. The test persons answered the question with an average of
The test persons disagree with the fact that the program was very awkward to use. The average answer to this question is 2.

<table>
<thead>
<tr>
<th>Test person</th>
<th>System Usability Scale result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>45</td>
</tr>
<tr>
<td>b</td>
<td>90</td>
</tr>
<tr>
<td>c</td>
<td>72,5</td>
</tr>
<tr>
<td>d</td>
<td>75</td>
</tr>
<tr>
<td>e</td>
<td>72,5</td>
</tr>
<tr>
<td>f</td>
<td>87,5</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>73,75</strong></td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td><strong>73,75</strong></td>
</tr>
</tbody>
</table>

Table 6.2: SUS result

Utility  The utility is measured with the result of the questions Q11 to Q17. Question Q11, Q12 and Q17 test a comparison of GTorXakis and TorXakis. Thereby Q11 tests if it is easier to model with GTorXakis. The test persons answered this question on average with 3.2 (median 4). Question Q12, if the modelling process is more clearly with GTorXakis, is answered with an average of 3.8 (median 4). If it is easier to test a SUT with GTorXakis is answered in Q17 with an average of 3.6 (median 4).

In question Q18 the test persons had to fill in positive or negative aspects of GTorXakis. The test person a had a problem to understand, which sense the textfields Gate and Action in the transition dialog have. Furthermore he/she thinks that it is not intuitive to create global variables to store some values. Test person b thinks that GTorXakis’s steering is very intuitive. Furthermore does b think that graphical models might not be usable if the SUT is very complex. Test persons c, d, e like the graphical model representation. Test person e thinks that through graphical models other people will understand a model faster. Furthermore test person c thinks that he/she still needs a textual input.
### Table 6.3: Questionnaire with results (1 = strongly disagree, 5 = strongly agree)

<table>
<thead>
<tr>
<th>ID</th>
<th>Questions</th>
<th>Median</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>I think that I would like to use this program frequently</td>
<td>4</td>
<td>3.5</td>
</tr>
<tr>
<td>Q2</td>
<td>I found the program unnecessarily complex</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q3</td>
<td>I thought the program was easy to use</td>
<td>4</td>
<td>3.67</td>
</tr>
<tr>
<td>Q4</td>
<td>I think that I would need the support of a technical person to be able to use this program</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Q5</td>
<td>I found the various functions in this program were well integrated</td>
<td>4.5</td>
<td>4.5</td>
</tr>
<tr>
<td>Q6</td>
<td>I thought there was too much inconsistency in this program</td>
<td>2</td>
<td>1.67</td>
</tr>
<tr>
<td>Q7</td>
<td>I would imagine that most people would learn to use this program very quickly</td>
<td>4</td>
<td>4.17</td>
</tr>
<tr>
<td>Q8</td>
<td>I found the program very awkward to use</td>
<td>1.5</td>
<td>2</td>
</tr>
<tr>
<td>Q9</td>
<td>I felt very confident using the program</td>
<td>4</td>
<td>3.5</td>
</tr>
<tr>
<td>Q10</td>
<td>I needed to learn a lot of things before I could get going with this program</td>
<td>3</td>
<td>2.67</td>
</tr>
<tr>
<td>Q11</td>
<td>It is easier to model with GTorXakis as with TorXakis</td>
<td>4</td>
<td>3.2</td>
</tr>
<tr>
<td>Q12</td>
<td>The modelling process of GTorXakis is more clearly</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>Q13</td>
<td>GTorXakis does not help at all</td>
<td>1</td>
<td>1.6</td>
</tr>
<tr>
<td>Q14</td>
<td>I would like to work with GTorXakis and not with TorXakis</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Q15</td>
<td>The Model Based Testing process is implemented in a good way</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>Q16</td>
<td>It is easy to change something in the model</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>Q17</td>
<td>It is easier to test a SUT with GTorXakis instead of TorXakis</td>
<td>4</td>
<td>3.6</td>
</tr>
</tbody>
</table>

### 6.3.2 Observation

In the following paragraphs the results of the observation by every Human Factor are outlined. Besides that, the results of the observation during the second exercise are summarized.

**Time to learn** In table 6.4 the time, the test persons needed to learn new things in GTorXakis, is shown. Therefore the tasks of the first exercise were structured, such that it is easy to measure how long the test persons need to perform given tasks for the first time. The test persons needed on average, for the first time, 36s to rename a model, 25s to add a state, 21s to add an edge, 28s to change a transition, 27s to add a gate and 17s to run a project. The implemented steering system of the drawing area, presented in section 3.2, is chosen because it is more dynamic. The experiment shows that the steering system is difficult to learn because it assumes that the user knows how the steering system works.
Table 6.4: Time to learn essential tasks (time in min)

<table>
<thead>
<tr>
<th>Task</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename a model</td>
<td>01:09</td>
<td>00:33</td>
<td>00:27</td>
<td>00:36</td>
<td>00:43</td>
<td>00:10</td>
<td>00:36</td>
</tr>
<tr>
<td>Add a state</td>
<td>00:42</td>
<td>00:05</td>
<td>00:53</td>
<td>00:13</td>
<td>00:17</td>
<td>00:21</td>
<td>00:25</td>
</tr>
<tr>
<td>Add an edge</td>
<td>00:20</td>
<td>00:05</td>
<td>01:15</td>
<td>00:04</td>
<td>00:11</td>
<td>00:12</td>
<td>00:21</td>
</tr>
<tr>
<td>Change transition</td>
<td>00:20</td>
<td>00:06</td>
<td>00:11</td>
<td>00:14</td>
<td>00:15</td>
<td>00:36</td>
<td>00:28</td>
</tr>
<tr>
<td>Add a gate</td>
<td>00:10</td>
<td>00:09</td>
<td>00:27</td>
<td>00:42</td>
<td>00:24</td>
<td>00:28</td>
<td>00:27</td>
</tr>
<tr>
<td>Run project with TorXakis</td>
<td>00:08</td>
<td>00:23</td>
<td>00:33</td>
<td>00:24</td>
<td>00:39</td>
<td>00:45</td>
<td>00:17</td>
</tr>
</tbody>
</table>

**Speed of performance**  The test persons needed 2 minutes and 36 seconds to add all gates and variables on average. Furthermore, they added all states on average in 27 seconds and all edges (+transitions) in 7:36 minutes. The editing process of edges (+transitions) differs a lot between the test persons. Test person c needed 15:40 minutes while test person d needed 04:52 minutes. This difference shows that the process of editing edges (+transitions) is not clear for every user. The task *Running the project in stepper mode* needed on average 2:35 minutes and *Running the project in tester mode* needed 1:07 minutes, where the test persons also started the SUT.

Table 6.5: Speed of performance (time in min)

<table>
<thead>
<tr>
<th>Task</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add gates and variables</td>
<td>01:55</td>
<td>02:19</td>
<td>03:08</td>
<td>03:33</td>
<td>03:47</td>
<td>00:58</td>
<td>02:36</td>
</tr>
<tr>
<td>Add all states</td>
<td>00:42</td>
<td>00:16</td>
<td>00:53</td>
<td>00:13</td>
<td>00:17</td>
<td>00:21</td>
<td>00:27</td>
</tr>
<tr>
<td>Add edges and transitions</td>
<td>05:35</td>
<td>06:19</td>
<td>15:40</td>
<td>04:52</td>
<td>07:39</td>
<td>05:31</td>
<td>07:36</td>
</tr>
<tr>
<td>Run stepper</td>
<td>00:36</td>
<td>00:34</td>
<td>00:33</td>
<td>00:24</td>
<td>00:39</td>
<td>00:45</td>
<td>00:35</td>
</tr>
<tr>
<td>Run tester (SUT)</td>
<td>01:15</td>
<td>01:52</td>
<td>00:54</td>
<td>00:45</td>
<td>01:00</td>
<td>01:01</td>
<td>01:07</td>
</tr>
<tr>
<td>Create sender model</td>
<td>-</td>
<td>-</td>
<td>06:39</td>
<td>04:07</td>
<td>07:00</td>
<td>09:46</td>
<td>06:53</td>
</tr>
<tr>
<td>Create communication model</td>
<td>-</td>
<td>-</td>
<td>01:47</td>
<td>01:38</td>
<td>01:54</td>
<td>02:05</td>
<td>01:51</td>
</tr>
</tbody>
</table>

**Errors** While analysing the screen and audio recordings, a few errors occurred during the experiment. Some test persons mentioned these errors at the end of the experiment. The renaming process of the models is not intu-
itive. In GTorXakis a model can be renamed through the menubar at the top of the program (see figure 6.3). The test persons stated that this action should be done with a double click on the model name and/or with a context menu that opens when the user clicks with a right mouse click. In GTorXakis the users can add edges between states, also if the edge starts and ends in the same state. The test persons often added these edges to the model, while they do not want to add these.

Furthermore some textfields in GTorXakis did not work as expected. For example in the sitebar shown in figure 4.6, where the gates and variables will be defined. The users expected that they can move through the textfields with the tabulator-key.

Almost every test person immediately started TorXakis but forgot to run the project with it. In GTorXakis the user needs to start TorXakis and then run the project with it.

**Modelling performance**  In the second exercise the test persons had to draw given models into GTorXakis, the sender model and a communication model of the Alternating Bit Protocol. In table 6.5 the measured time is shown.

The sender model of the Alternating Bit Protocol, that contains three states and three edges is created on average in 6 minutes and 53 seconds. On average the test persons needed for every action 01:05 minutes.
The second model, the communication channel, consists of two states with two edges. This model is created in 1 minute and 51 seconds on average. In this model, every action takes on average 00:27 minutes. On page 49 we say that the steering system is difficult to learn. The modelling performance shows, if users know how the steering system works, that models can be created fast.
6.4 Validity threats

In this experiment the test persons are chosen from TorXakis users. Due to the fact that there are not many TorXakis users at this moment, the results of the experiment could be biased. For example test person d is a developer of TorXakis, which means that he/she could rate TorXakis in the comparison with GTorXakis better.

In spite of that the validity of this research is given, because at the moment GTorXakis needs the knowledge of TorXakis which do not have many people. In the future the same experiment could be done with users who are new to TorXakis and GTorXakis.

This first prototype and usability study of GTorXakis does show what part works and what can be improved.
Conclusion and discussion

In this chapter the results from the previous chapter are analysed and conclusions will be drawn. With help of these conclusions the research question will be answered. Furthermore the conclusions are discussed.

7.1 Conclusion

In this section all sub-questions of the research question will be answered. At the end the research question itself is answered.

What does the Model Based Testing process look like?

In section 2.3 the Model Based Testing process is explained. In the experiment the test persons were asked to describe the Model Based Testing process in their own words. The Model Based Testing process of test person b adds the retrieval process of the interface/part that needs to be tested. In figure 7.1 this retrieval process is added. Therefore the Model Based Testing process discussed in section 2.3 is taken as a basis. Furthermore a possible discussion with the developers could be taken as a basis for creating the model.

The other test persons did not describe the MBT-process but rather the modelling process. In figure 6.1 and figure 6.2 such processes are shown. These processes will describe in detail the Create/Change model part of the
Model Based Testing process constituted in section 2.3.

![Figure 7.1: MBT-process with changes of test person b](image)

**How does *TorXakis* support the Model Based Testing process?**

*TorXakis* supports the test generation and test execution of the Model Based Testing process shown in figure 2.10. In some cases *TorXakis* also supports the SUT, if it is simulated.

Test person d described the MBT-process with *TorXakis* as a modelling tool. In figure 6.2 this process is represented. In this process the SUT is modelled while checking and verifying it with *TorXakis*’s stepper mode. After the model is checked, the SUT is tested with the tester mode of *TorXakis*.

The MBT-process shown in figure 2.10 does not separate these two options of *TorXakis*, because both options will use the test generation and execution of *TorXakis* and can be combined to one global test generation, -execution. The modelling process, described by test persons a and f, is not supported by *TorXakis* itself.
Which parts of the Model Based Testing process need to be implemented in the Graphical User Interface?

In figure 2.10 the parts, which GTorXakis needs to implement, are shown. The core task of GTorXakis is the modelling part. As the test persons a and f show, the Graphical User Interface needs to fully support the following actions:

- Adding states
- Connect states with transitions
- Adding gates and variables to a model
- Run the project with TorXakis in every possible mode (e.g. Stepper)
- Possibility to define all kind of type definitions that TorXakis supports

These actions are fully supported by GTorXakis. In the newly created drawing area, the users can add states, transitions between states and gates and variables to models. In addition to that it should be easy to change a model, e.g. if errors in the model were found.

Which design guidelines can be implemented?

In section 2.5 different design guidelines were discussed. All of the presented examples of the guidelines were tried to implement in GTorXakis. In table 7.1 the success of these implementations were shown. For example the usage of icons, appropriate feedback and the fact that the knowledge about the old tool TorXakis is unnecessary. In GTorXakis the user should always get feedback for actions which he/she does. If the user adds a state, edge or comment, everything is represented on the screen. Furthermore users will see the result of TorXakis in different colors, so that they can easily recognize the result of their test run.
Table 7.1: Design requirements with results

<table>
<thead>
<tr>
<th>Requirement</th>
<th>GTorXakis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always appropriate feedback</td>
<td>X</td>
</tr>
<tr>
<td>Undo-/Redoing from actions</td>
<td>X</td>
</tr>
<tr>
<td>Use of icons</td>
<td>X</td>
</tr>
<tr>
<td>Background color lighter</td>
<td>X</td>
</tr>
<tr>
<td>Foreground color darker</td>
<td>X</td>
</tr>
<tr>
<td>Consistent</td>
<td></td>
</tr>
<tr>
<td>Knowledge about TorXakis should not be redundant</td>
<td>X</td>
</tr>
</tbody>
</table>

The syntax of the TorXakis input language has not changed in GTorXakis, although it is not superficial. It has not changed because of an important guideline of user interface design, namely that the knowledge about TorXakis should not be redundant. GTorXakis supports all type definitions that TorXakis supports.

Another important detail in GTorXakis is the steering system of the drawing area. In section 3.2 the new steering system of GTorXakis is represented. The steering system of GTorXakis is taken from an other tool, yEd, that is used in MBT by Graphwalker. GTorXakis is not consistent in every situation. The renaming process is not intuitive and not in line with the renaming process of states and edges.

Is the new Graphical User Interface usable for users of TorXakis?

The new Graphical User Interface, GTorXakis, is usable for TorXakis users. The results of the System Usability Scale vary between a minimum of 45 and a maximum of 90. Both values are the results of the test persons who were only able to do the first exercise.

The average and median result of the System Usability Scale of all test persons are 73,75 (Table 6.2). On a scale between 0 and 100 this value can be
interpreted as an acceptable score. The variation of the values is acceptable too, because the median and average is the same. The usability of systems also vary from user to user, because of the personal feeling of users.

From the results of the utility questions, the test persons can not estimate if it is easier to model with GTorXakis. It might be, that they need more time to give an answer to this question or it might be, that the old syntax of the TorXakis-input language is also assessed with this question.

According to the questions Q12 and Q17 the test persons agree, that the modelling process with GTorXakis is clearer and that it is easier to test a SUT with GTorXakis. The personal feelings from question Q18 show, that some test persons like GTorXakis and some of them like TorXakis more.

In table 6.4 the time, that the test persons needed to perform given tasks for the first time, is shown. In comparison with the time, they needed to fully create a model in the second exercise (Paragraph 6.3.2), the users were able to perform this tasks faster. For the first model the users needed on average 01:05 minutes and in the second model, 00:27 minutes for an action. An action might be for example: adding state, adding edge, adding gates, etc.. Nevertheless the measured time can be improved in some cases. For example the renaming process of a model, according to the test persons, is not intuitive. In addition to that the errors, presented in section 6.3.2, need to be fixed.

The next indicator of the usability from GTorXakis is the speed of performance. Following the observations the mechanism to add new states is mostly independent from the user itself, because the time does not differ too much between the test persons. The addition of edges and transitions in GTorXakis is not very intuitive, because the measured time is relatively high. This is promoted by the errors that can be seen in the screen recordings. The time varies between 04:52 minutes and 15:40 minutes.

In comparison to that, the users were able to run the project fast in stepper and tester mode. Nevertheless this can be improved too. To run a project
with TorXakis the test persons needed to start and run it manually by pressing two buttons. Both actions can be triggered by only one button.

In summary, it can be stated that the usability of GTorXakis is good at the moment, but improvable through the constituted errors and the things the test persons wrote down in question Q18.

**How can the Model Based Testing process of TorXakis be improved by designing a Graphical User Interface for it?**

In this research the Model Based Testing process was analysed and a new Graphical User Interface was developed to test if the Model Based Testing process can be improved. In section 1 the current situation and a possible solution was outlined.

In the current situation (see section 1.1) graphical models are not created in TorXakis. If users want a graphical model, the model needs to be drawn in another application. The model needs to be transformed to the TorXakis-input language, if the users want to use it in TorXakis. If the model needs to be adapted, both models need to be updated manually.

GTorXakis solves this problem by converting graphical models to the TorXakis-input language automatically. The Model Based Testing process is improved by drawing one model instead of drawing a model and manually transform it to the TorXakis-input language.

The usability of GTorXakis is another important factor by analysing if the Model Based Testing process can be improved, because a bad usability will probably extend the process. The usability of GTorXakis is good, but can be improved by fixing some errors and making improvements that are presented in the previous answer.

In summary, one could say that the Model Based Testing process of TorXakis can be improved by creating a new Graphical User Interface (GTorXakis).
It makes the modelling process more clearly and it is easier to test a SUT, which is definitely an improvement of the Model Based Testing process.

7.2 Discussion

In a future research the usability should be validated again. The errors during the experiment and the suggestions of the test persons need to be implemented in a new version of GTorXakis. After that, the usability should be validated again with the System Usability Scale and the Human Factors. Thereby the Human Factor Retention over time should be analysed too. The results of this research might be validated with the future research while comparing the results of the System Usability Scale and the Human Factors.

Furthermore an other setup of the experiment might be interesting. An example might be to use the program in the course Testing Techniques at the Radboud University and check what new users of TorXakis think about GTorXakis and it’s steering system.

It might be a good idea to look at other Model Based Testing tools and check if there are functions that can be included in GTorXakis. For example has JTorX\textsuperscript{1} the opportunity to represent the test logs graphically.

The use of the programming language Java and the Batik library was a good choice. The library has much power and offers a good functionality to build a canvas where states and edges can be added. Anyway Batik requires deep knowledge about SVG, and that is why people without knowledge about Batik should learn SVG first.

GTorXakis is developed really fast and some functionality, e.g. renaming of models, need to be fixed before releasing it. It should be able to create an installer such that all the dependencies of TorXakis and GTorXakis are installed automatically.

\textsuperscript{1}https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/
Bibliography


Appendix

Model Based Testing - GTorXakis

Thanks for helping me with my masterthesis. Please answer the following questions carefully! You are free to answer questions 18 and 19 in English, Dutch or German.

Usability

1 I think that I would like to use this program frequently
   Strongly disagree □—□—□—□—□ Strongly agree

2 I found the program unnecessarily complex
   Strongly disagree □—□—□—□—□ Strongly agree

3 I thought the program was easy to use
   Strongly disagree □—□—□—□—□ Strongly agree

4 I think that I would need the support of a technical person to be able to use this program
   Strongly disagree □—□—□—□—□ Strongly agree

5 I found the various functions in this program were well integrated

63
6 I thought there was too much inconsistency in this program
   Strongly disagree □—□—□—□—□ Strongly agree

7 I would imagine that most MBT people would learn to use
   this program very quickly
   Strongly disagree □—□—□—□—□ Strongly agree

8 I found the program very awkward to use
   Strongly disagree □—□—□—□—□ Strongly agree

9 I felt very confident using the program
   Strongly disagree □—□—□—□—□ Strongly agree

10 I needed to learn a lot of things before I could get going with
    this program
    Strongly disagree □—□—□—□—□ Strongly agree

Comparison with *TorXakis*

11 It is easier to model with GTorXakis as with *TorXakis*
   Strongly disagree □—□—□—□—□ Strongly agree

12 The modelling process of GTorXakis is more clearly
   Strongly disagree □—□—□—□—□ Strongly agree

13 GTorXakis does not help at all
   Strongly disagree □—□—□—□—□ Strongly agree

14 I would like to work with GTorXakis and not with *TorXakis*
   Strongly disagree □—□—□—□—□ Strongly agree

15 The Model Based Testing process is implemented in a good
    way
   Strongly disagree □—□—□—□—□ Strongly agree
16 It is easier to change something in the model
Strongly disagree □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□®
1 Adder

In this exercise you create your first model with GTorXakis. GTorXakis is a Graphical User Interface that connects to TorXakis with a socket connection. In GTorXakis the user is able to create graphical model representations. These representations are later casted as state automaton (STAUTDEF) in TorXakis. In this exercise you only need to implement the graphical model.

1.1 Theoretical background

The adder is a simple SUT that expects the following inputs:

- \texttt{Plus \ ?x \ ?y}
- \texttt{Minus \ ?x \ ?y}

where \( x, y \in \mathbb{N} \)

With these inputs the SUT calculates the belonging result. For example if the SUT gets the input \texttt{Plus 3 5} it calculates \( 3 + 5 = 8 \). The output of the SUT is the result, in our example it sends 8. If the input starts with \texttt{Minus}, then the SUT calculates the subtraction of the following two integers.

1.2 Exercise

\textbf{Task 1}  Read the already defined TYPEDEF, MODELDEF and CNECTDEF

\textbf{Task 2}  Rename the graphical model to \textit{adder}

\textbf{Task 3}  Add all the gates and variables that are needed in the model

(The order in "adder[Action, Result]()" needs to be the same as in the new model.)

\textbf{Task 4}  Add necessary states to the model

\textbf{Task 5}  Add necessary edges between those states

(In GTorXakis an edge between two states represents all edges between those states.)

\textbf{Task 6}  Change the transitions of these edges (To check if an Operation \textit{opn} is Plus or Minus, you can simply call the functions isPlus(\textit{opn}) and isMinus(\textit{opn}).)

\textbf{Task 7}  Run your program in Stepper mode with 100 steps
Task 8  If everything worked correctly in step seven, then start the SUT and run your program in Tester mode with 200 steps
How to start the SUT:

- Press *Windows + R*
- Enter *cmd*
- Enter *cd Desktop*
- Enter *java -jar Adder.jar 7890*
2 Alternating Bit Protocol

In this exercise you will build the whole project file on your own. Furthermore you need to simulate the SUT with another instance of GTorXakis. You can easily use the same models, but you will need another CNECTDEF that is the opposite of the other one.

2.1 Theoretical background

The following definition of the Alternating Bit Protocol is taken from Wikipedia. A simplified model of the Alternating Bit Protocol is shown in figure 1.

**Definition**

Alternating Bit Protocol (ABP) is a simple network protocol operating at the data link layer that retransmits lost or corrupted messages. It can be seen as a special case of a sliding window protocol where a simple timer restricts the order of messages to ensure receivers send messages in turn while using a window of 1 bit.

Messages are sent from transmitter A to receiver B. Assume that the channel from A to B is initialized and that there are no messages in transit. Each message from A to B contains a data part and a one-bit sequence number, i.e., a value that is 0 or 1. B has two acknowledge codes that it can send to A: ACK0 and ACK1.

When A sends a message, it resends it continuously, with the same sequence number, until it receives an acknowledgment from B that contains the same sequence number. When that happens, A complements (flips) the sequence number and starts transmitting the next message.

When B receives a message that is not corrupted and has sequence number 0, it starts sending ACK0 and keeps doing so until it receives a valid message with number 1. Then it starts sending ACK1, etc.

This means that A may still receive ACK0 when it is already transmitting messages with sequence number one. (And vice versa.) It treats such messages as negative-acknowledge codes (NAKs). The simplest behaviour is to ignore them all and continue transmitting.

![Figure 1: Alternating Bit Protocol](image)

Experiment № 1
2.2 Exercise

In the exercise we will assume that every message will be received by the receiver. Thus every message will be send exactly one time.

**Task 1** Create the sender model

![Sender Diagram](image)

(Hint: Do not forget to add the gates and variables.)

**Task 2** Create the communication channel: Sender -> Receiver (Figure 3)

![Communication Channel: Sender -> Receiver Diagram](image)

**Task 3** Create the communication channel: Receiver -> Sender

**Task 4** Create the receiver model

**Task 5** Read the PROCDEF that connects all the models
(Do not forget to check the order of the gates.)

**Task 6** Test the result in Stepper mode with the CNECTDEF of the SUT
Task 7  Open the project in a second instance of GTorXakis

Task 8  Start the simulator in the first instance

Task 9  Start the tester in the second instance
(Hint: Do not use the same port for two instances of TorXakis.)

Task 10  Run the simulator

Task 11  Run the tester with 100 steps