Radboud University Nijmegen

Model-based testing Smart Cable Guard, an embedded system

Master Thesis
Software Science

Author:
G.P. Noordbruis
Gunnar.Noordbruis@dnv.com
Gunnar.Noordbruis@ru.nl

DNV supervisor:
dr. P. Wagenaars
Paul.Wagenaars@dnv.com

Radboud supervisor & First Assessor:
dr. ir. G.J. Tretmans
Jan.Tretmans@tno.nl
Jan.Tretmans@ru.nl

Second Assessor:
dr. F.W. Vaandrager
Frits.Vaandrager@ru.nl

September 2, 2022

Acknowledgements

First and foremost, I am extremely grateful to Jan Tretmans, Gijs van Cuyck and Paul Wagenaars for their advice, feedback, and patience during my master thesis. My gratitude extends to Jurgen de Bruijne, Richard van Harten and Bart Kruizinga for their help with SCG. Additionally, I would like to thank Pieter Koopman for his feedback and guidance during my internship, Richard Denissen for the opportunities at DNV and Frits Vaandrager for his willingness to aid in the final assessment of this thesis. I would like to thank all the members of the SCG team.
Also, I would like to thank Axini for allowing me to use their model-based testing tool and for providing support. Finally, I would like to express my deepest gratitude to my parents, brothers and girlfriend. Without their tremendous support, understanding and encouragement in the past years, it would have been an impossible journey.

Abstract

Smart Cable Guard (SCG) is an embedded system that protects medium voltage power cables all over the world using Control Units (CUs). These CUs perform measurements to detect and locate problems within the cable. The current testing strategy to check critical functions is costly and time consuming. Therefore, the SCG team is looking to apply Model-Based Testing (MBT) to SCG. This thesis presents a comparison between manual, automated and model-based testing of SCG, and between the two applied testing tools. Alongside that, a generalization of applying MBT to embedded systems is discussed. Most importantly, this thesis explores applying MBT to SCG to find defects and to increase software quality.

The System Under Test (SUT) and test harness used for MBT are constructed. Two MBT tools are applied on the SUT. Then, manual, automated and model-based testing are compared. Additionally, the two testing tools are compared. Based on the literature and the approach of applying MBT to SCG, a generalized approach for applying MBT to embedded systems was created.

The application of MBT to SCG resulted in two proof of concept implementations, which found three software defects. The implementations are a replacement for 14 out of the 27 system level tests. Unlike the system tests, MBT takes into account the timing requirements. In addition, MBT performs several scenarios concurrently and for a longer period of time. Furthermore, the generalization of MBT on embedded systems resulted in a new hardware layer in the test harness. Applying MBT to SCG on a system level had the advantages that it does not require any test doubles. In some cases additional interfaces are needed to apply MBT.

To conclude, it was shown how to apply MBT to SCG. MBT can contribute to the testing of SCG by reducing manual effort and increasing the test coverage. Both MBT tools, TorXakis and Axini, have points of improvement. However, Axini is more refined than TorXakis. It is difficult to generalize MBT for embedded systems. The SUT needs to be carefully chosen. The testing of embedded generally requires an additional hardware layer in the test harness.

Contents

Glossary

ADT Algebraic Data Type. 12, 18, 25, 26, 41, 52, 54

AMP Axini Modeling Platform. 16, 17

CFE Communication Front-End. 5

CU Control Unit. 1, 4–6, 21–23, 25, 34–37, 39, 44–48, 60

GPIO General Purpose Input/Output. 23

IOCO Input Output Conformance. 3, 9–12, 26, 28, 57, 59, 62

IUT Implementation Under Test. 9

LTS Labeled Transition System. 3, 6–9, 43, 57

MBT Model-Based Testing. 1–3, 6, 9–12, 16, 21, 22, 44–46, 48–50, 52, 53, 60–65

MQTT MQ Telemetry Transport. 6, 22, 23, 25, 26, 39

PD Partial Discharge. 3, 6, 22–24, 37, 47

RTIOCO Relativized Timed Input Output Conformance. 62

SCG Smart Cable Guard. 1–5, 21, 22, 24, 38, 41, 43–50, 60, 63, 64

SIOCO Symbolic Input Output Conformance. 57

SMT Satisfiability Modulo Theories. 58, 59

STS Symbolic Transition System. 57

SU Sensor Unit. 1, 5, 6, 21–23, 44–47, 60

SUT System Under Test. 1, 2, 8–11, 13–17, 19–23, 25, 27, 28, 32, 35–37, 39, 40, 43, 44, 49, 50, 53, 58, 60–62, 64

TCP Transmission Control Protocol. 6, 15, 16

TIOCO Timed Input Output Conformance. 62

TOCTOU time-of-check time-of-use. 34, 37

TTL Transistor-Transistor Logic. 22–24

TXS TorXakis Syntax. 24

Chapter 1

Introduction

The dependence on electricity is high and still increasing with electricity replacing more and more fossil fuels. This electrification demands a very stable electrical network. The increasing demand is also putting many networks at their maximum capacity. Additionally, more homes are being equipped with solar panels causing periodic surges on the net [1]. For the first time network operators in Northern Holland are allowed to utilize more than 50% of the capacity. The other 50% is strictly reserved for maintenance and outages[2].
In recent years more network operators have indicated that their networks are being overloaded[3]. Cables that are being overloaded have a higher chance of failure. If upcoming failure can be predicted, adequate measures can be taken before power outages occur.
To predicted upcoming failure, monitoring of the electricity network is possible using Smart Cable Guard (SCG). SCG in total is a sensor based electricity cable monitoring platform. SCG uses a brain called the Control Units (CUs) and sensors called Sensor Unit (SU) to monitor a cable. The SUs detect and locate both weak spots in the cable and faults (e.g. earth faults). To determine the location of a problem two sensors are required per cable.
Accurate around the clock monitoring requires a high degree of reliability of the hardware and software of device in the field. The devices in the field are what can be seen as a brain (CUs) and multiple sensors . During the lifetime of SCG several software defects have shown that the current testing strategy is insufficient. The strategy does not provide adequate confidence in the reliability of SCG. The test strategy consists of unit tests combined with manual verification of several system requirements on an actual CU. While unit tests are an essential part to ensure software quality [1]. Unit tests only test a very small part of the system. Furthermore, manual verification is error

[1] nos.nl/l/2390461
[2] nos.nl/l/m/2298022
[3] volkskrant.nl/cs-bd07d87a

prone and tedious work. SCG wishes to increase the confidence in the reliability of the devices. This can potentially be done using simple automated testing or by applying Model-Based Testing (MBT) to the devices. Both options are being considered, this thesis investigate the MBT possibilities and to guide the SCG team in the next steps in test automation. The focus in this thesis is on the devices, since they are deployed in the field, where physical access is infrequent. It is costly and not trivial to service these devices in case of a software problem.

The MBT tools TorXakis[4] and Axini[5] will be applied to the SCG system. The MBT tools have different feature sets that could make one tool more suitable for the System Under Test (SUT). Axini is capable of modeling time constraints while TorXakis is not. Additionally, there is ongoing research and development with both tools, which is a requirement of possible usage at DNV. The desire to realize higher software reliability lead to the research questions handled in this thesis.

1. How can model-based testing be applied to the Smart Cable Guard system to find defects and to increase software quality?
2. How does model-based and automated testing compare to each other and to the manual system testing currently applied to SCG?
3. How do TorXakis and Axini compare on the aspects of their modeling language, communication functions, error messages, documentation, visualization, conformance theory and constraint solving?
4. How can the experience of model-based testing SCG be generalized to other embedded systems?

In this thesis, chapter 2 provides the preliminary information related to the SCG system, the theory behind MBT, basic information of MBT and finally the syntax of the two MBT tools used. Chapter 3 goes into detail on the chosen SUT, the test harness used for MBT, the approaches for both MBT tools and the found issues. Chapter 4 discusses the current testing strategy and compares it to the model-based testing of SCG presented in chapter 3. Chapter 5 presents a comparison of TorXakis and Axini base on their modeling language, functions, communication, error messages and documentation. Chapter 6 discusses the generalization of the approach taken in chapter 3. In particular, defining the SUT and creating an appropriate test harness for an embedded system are discussed.

[4] torxakis.org
[5] axini.com

Chapter 2

Preliminaries

In this chapter the needed background for this thesis will be discussed. This includes basic information regarding problems in electricity cables and high level information of the SCG system. MBT will be applied to SCG in this thesis. As such, this chapter introduces the modeling formalisms underpinning the formal specifications and the models of implementations used in MBT.
Labeled Transition Systems (LTSs) are the basic model used by the conformance relation Input Output Conformance (IOCO). Additionally, LTSs are often used in model checking [2] and model-based testing [3]. They are introduced in section 2.3. Furthermore, in section 2.4 the conformance relation IOCO is presented. IOCO specifies when an implementation conforms to the provided model. Besides this, the notion of model based testing will be discussed in section 2.5. Finally, the workings and syntax of the two MBT tools used in this thesis will be explained in sections 2.6 and 2.7 respectively.

2.1 Faults and Partial Discharge

SCG looks for faults like earth faults or short-circuit faults and Partial Discharge (PD). There is an important distinction between faults and PD. PD does not directly cause failure of the cable. PD is a very small discharge occurring in a void within the insulation of a cable. Figure 2.1 shows where PD takes place within a cable. The small discharge bridges the void in the insulation. It will slowly erode the insulator. The voids are usually impurities within the insulator. It is hard to discover as the amplitude of the charge is very small compared to a fault. Faults have a large amplitude and they usually occur when a cable breaks down. Due to their large amplitude faults are not difficult to detect. Detecting and locating both faults and PD is useful to perform quick and precise repairs.

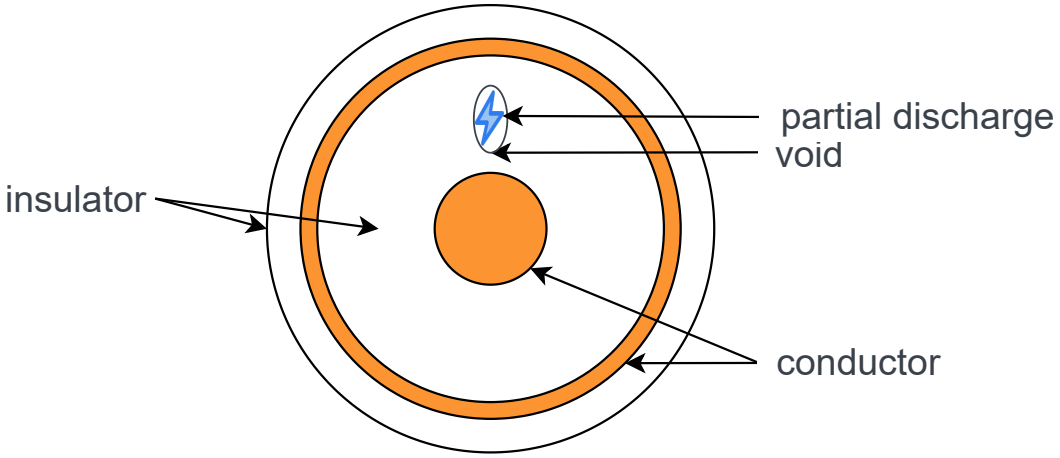

Figure 2.1: A partial discharge shown within a void in a section of cable.

2.2 Smart Cable Guard

Smart Cable Guard is a sensor-based system that monitors medium voltage cables to detect and locate faults and weak spots in underground cables. This is done by sensors, called Sensor Units (SUs), placed around the cable at two ends. The system is accompanied by a backend for analysis, processing and viewing of data. The current version of SCG has been under development since 2015. Figure 2.2 displays how a measurement system is deployed in the field. On either side of the cable the sensors are placed around the cable. The houses are secondary substations or in Dutch "transformatorhuisjes". The black sections in each substation is a joint, connecting the cable to another. There is no connection between the two Control Units (CUs) other than the electricity cable. The CUs are not part of the backend but merely submit data to the backend using their mobile network connection.

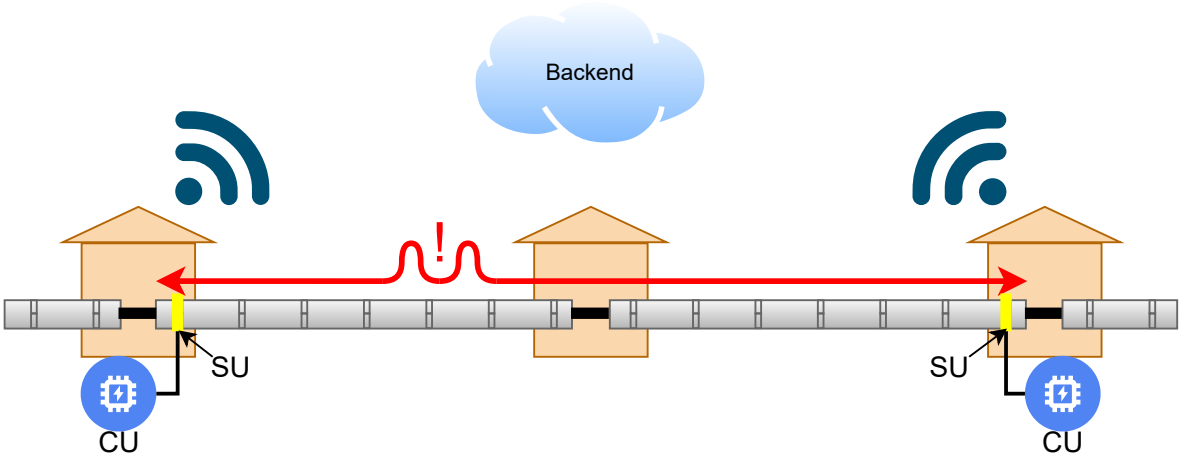

Figure 2.2: Depiction of two CUs and their SU installed in the field. The maximum length between the two SUs is 15Km.

A more detailed depiction of the backend can be seen in figure 2.3. In this figure, the SUs and the cables between the CUs are left out. The Communication Front-End (CFE) in the middle provides the system with communication capabilities. If a CU loses the capability to communicate or perform commands, a technician needs to visit the unit in the field to either replace it or perform manual reset steps. This is a time consuming and costly endeavour. In the past the SCG team has seen several cases where the software caused such failures. Due to this, increasing the reliability of a CU through thorough testing is desired.

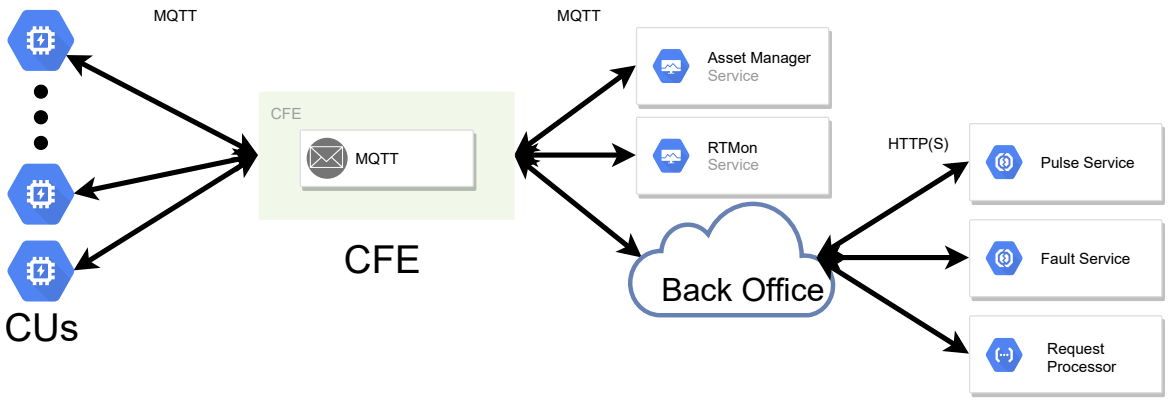

Figure 2.3: SCG platform design

In figure 2.3 the CUs are depicted in a simple manner. However, each monitoring system is more complex. As already discussed each monitoring system is made up out of at least two CUs and at least one SU each. Figure 2.2 shows how the monitoring systems are installed with a live electricity cable going through the SUs. One CU has multiple in

and outputs. The SU provides the CU with fault and PD information. Additionally, the SUs establish synchronization with each other over the medium voltage power cable. The synchronisation is required to locate faults and PDs. In addition, the CU can be managed locally or remotely. The remote interface uses MQ Telemetry Transport (MQTT), which is a communication protocol on top of Transmission Control Protocol (TCP). These two have some overlap in terms of configuration options. Furthermore, the CU has 4 buttons, 4 LEDs and input power. The buttons are used to attempt auto synchronization for the different SUs and restart the CU. This can also be performed through the remote interface. The LEDs indicate the state of the SUs and if the CU is connected with the backend. The input power is considered an input because the CU is often powered by the cable being monitored. In the case of a fault it is useful to know if the power also went down. All these different in and outputs are shown in figure 2.4.

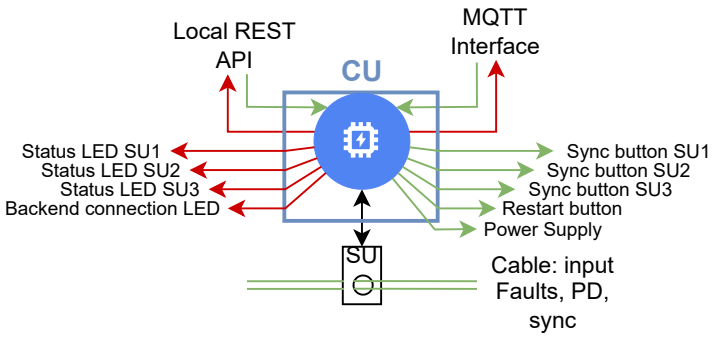

Figure 2.4: A single CU and SU with all in and ouputs.

2.3 Labeled Transition Systems

In this section, the relevant theory of Labeled Transition systems is introduced. They are used as a the underlying structure for MBT. LTSs are a combination of states, labels and transitions from a state to a state. The transitions are accompanied by a label. In light of MBT, these labels represent different actions. The structure of an LTS can vary. A basic LTS is a triple of the above mentioned parts [4, 5]. In other cases an LTS is a 4-tuple adding an initial state [6, 7] this is called a rooted LTS [4]. Furthermore there is a 5-tuple that additionally separates Input and Output labels [8]. Below the 4 and 5 tuple LTS will be discussed as shown in [6, 7, 8] by Tretmans, Stoelinga et al. and Tretmans et al.

Definition 1. *An LTS is a 4-tuple* $\langle S, s_0, L, T \rangle$ *where*

- S *is a countable, non-empty set of states;*
- $s_0 \in S$ *is the initial state;*
- L *is a countable set of labels;*

- $T \subseteq S \times (L \cup \{\tau\}) \times S$, *with* $\tau \notin L$, *is the transition relation;*

$s \xrightarrow{\ell} s'$ denotes the transition from state s to state s' using the label ℓ. Formally this is denoted as $(s, \ell, s') \in T$. Informally, a transition can be thought of as the system being in state s performing action ℓ and through that arriving in state s'.

Definition 2. *Considering an LTS* $A = \langle S, s_0, L, T \rangle$ *and two states* $s, s' \in S$.
A path of length n from s to s' *is a sequence of n transitions* $s \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \ \ldots \ s_{n-1} \xrightarrow{\ell_n} s'$ *where* $\forall_n^{1..n} : \ell_n \in (L \cup \{\tau\})$ *If* $n = 0$ *the path is empty and* $s = s'$. *Such a path can also be written as:* $s \xrightarrow{\ell_1 \cdot \ldots \cdot \ell_n} s'$.

$$s \xrightarrow{\ell_1} s' \Leftrightarrow_{def} (s, \ell_1, s') \in T$$
$$s \xrightarrow{\ell_1 \cdot \ldots \cdot \ell_n} \Leftrightarrow_{def} \exists s' : s \xrightarrow{\ell_1 \cdot \ldots \cdot \ell_n} s'$$
$$s \xrightarrow{\ell_1 \cdot \ldots \cdot \ell_n} \!\!\!\!\!\!/ \ \Leftrightarrow_{def} \neg \exists s' : s \xrightarrow{\ell_1 \cdot \ldots \cdot \ell_n} s'$$

These transitions can be composed. If two transitions exist such that $s \xrightarrow{\ell} s'$ and $s' \xrightarrow{\ell'} s''$, the composition can be written as $s \xrightarrow{\ell \cdot \ell'} s''$. The labels in L are observable actions that a system can perform. This set does not include τ. τ is a special transition label indicating an unobservable internal step. If $a \cdot \tau \cdot b \cdot \tau \cdot \tau \cdot c$ is a sequence of actions from s to s' ($s \xrightarrow{a \cdot \tau \cdot b \cdot \tau \cdot \tau \cdot c} s'$) then $s \xRightarrow{a \cdot b \cdot \cdot c} s'$ is the sequence of observable actions from s to s'. Let sigma be a sequence of actions such that a system being in a state s may reach state s' by performing all actions in σ. It is possible that it does not arrive in s' due to nondeterminism. The function $traces(s)$ denotes all possible sequences of observable actions starting in the provided state s. Traces can be defined as $traces(s) = \{\sigma \in L^* | s \stackrel{\sigma}{\Rightarrow}\}$ with their corresponding label set L. Functions defined on a state can be applied to LTS by applying them on the initial state. As such, for a LTS $TS = \langle S, ts_0, L, T \rangle$: $traces(TS) = \{\sigma \in L^* | ts_0 \stackrel{\sigma}{\Rightarrow}\}$

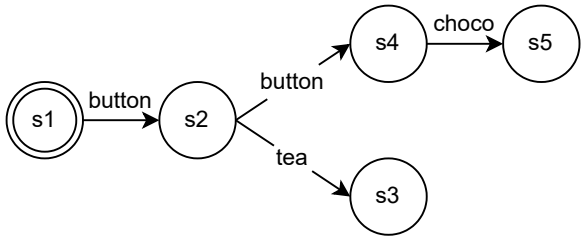

Figure 2.5: example graph representing LTS G with the double circle in state s_1 indicating the initial state.

$$\begin{aligned} G =& \langle \{s_1, s_2, s_3, s_4, s_5\}, s_1, \{button, tea, choco\}, \\ & \{(s_1, button, s_2), (s_2, tea, s_3), (s_2, button, s_4), (s_4, choco, s_5)\} \rangle \\ traces(G) =& \{\epsilon, button, button \cdot tea, button \cdot button, button \cdot button \cdot choco\} \end{aligned}$$

In the LTS in figure 2.5, it can be seen that no distinction has been made based on input and output labels. All labels are collected in one set. However, on closer inspection the label 'button' can be considered input while tea and choco are usually considered outputs of the SUT.

Definition 3. *A LTS with separate input and output labels is a 5-tuple* $\langle S, s_0, I, O, T\rangle$ *where*

- S *is a countable, non-empty set of states;*
- $s_0 \in S$ *is the initial state;*
- I *and* O *are countable sets of input and output labels respectively; where* $L = I \cup O$ *and* $\emptyset = I \cap O$
- $T \subseteq S \times (I \cup O \cup \{\tau\}) \times S$*, with* $\tau \notin (I \cup O)$*, is the transition relation;*

To distinguish the in and output labels in representations like figure 2.6, question and exclamation marks are used in front of labels for input and output, respectively.
The class of all LTSs over a given Input and Output label set $\mathcal{I}$ and $\mathcal{O}$ is represented by $\mathcal{LTS}(\mathcal{I}, \mathcal{O})$. A state that lacks a transition labeled with τ is called stable. Furthermore, a state lacking both an internal and an output action is called quiescent, which is denoted by the symbol δ ($\delta \notin (I \cup O \cup \tau)$). The shorthand notion L_δ is used for $L \cup \delta$. The transition $s \xrightarrow{\delta} s$ represents the absence of any transition $s \xrightarrow{\ell}$ where $\ell \in (O \cup \{\tau\})$. In figure 2.6 states s_1, s_3 and s_5 are quiescent.

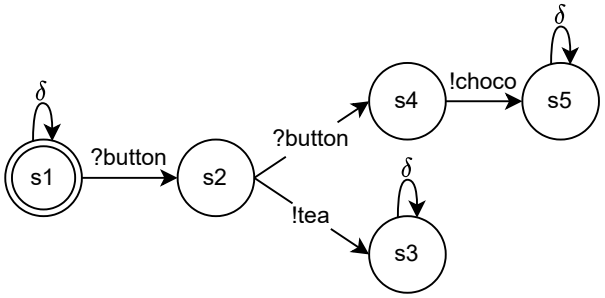

Figure 2.6: Example LTS S with marked input and output labels

$$G = \langle \{s_1, s_2, s_3, s_4, s_5\}, s_1, \{?button\}, \{!tea, !choco\}, \\ \{(s_1, ?button, s_2), (s_2, !tea, s_3), (s_2, ?button, s_4), (s_4, !choco, s_5)\}\rangle$$

The function *traces* does not include the above introduced quiescence. These can be captured with suspension traces. Suspension traces are the traces over the set L_δ. They are defined as $Straces(s) =_{def} \{\sigma \in L_\delta^* | s \stackrel{\sigma}{\Rightarrow}\}$. Besides this, for conformance relations it is important to know which states are reachable using a given suspension trace. This can be defined as TS **after** $\sigma = \{s' | ts_0 \stackrel{\sigma}{\Rightarrow} s'\}$ where $TS \in \mathcal{LTS}(I, O)$, ts_0 is the initial

state of TS and $\sigma \in L^*_\delta$. Moreover, the function $out(s)$ denotes all observable outputs from a given state s. It is defined as: $out(s) = \{\ell \in O | s \xrightarrow{\ell}\} \cup \{\delta | s \xrightarrow{\delta}\}$. When out is applied to a set of states it is defined as $out(S) = \bigcup\{out(s) | s \in S\}$. Bellow there are some examples of `after` and `out` for the LTS S displayed in figure 2.6.

$$\begin{aligned} S\ after\ \epsilon &= \{s_1\} \\ S\ after\ ?button &= \{s_2\} \\ S\ after\ ?button{\cdot}?button &= \{s_4\} \\ S\ after\ ?button{\cdot}!tea &= \{s_3\} \\ out(S) &= \{\delta\} \\ out(s_2) &= \{!tea\} \\ out(S\ after\ ?button) &= \{!tea\} \end{aligned}$$

An LTS is input enabled when every state has an outgoing transition for every input. The class of input enabled LTSs is denoted as $\mathcal{IOTS}(\mathcal{I}, \mathcal{O})$. For this class it holds that for all input and output label sets $\mathcal{I}$ and $\mathcal{O}$: $\mathcal{IOTS}(\mathcal{I}, \mathcal{O}) \subseteq \mathcal{LTS}(\mathcal{I}, \mathcal{O})$

2.4 Input Output Conformance Relation

In the previous section the basic models on which IOCO is defined are introduced. In this section, the IOCO theory will be explained and in the next section the notion of MBT will be presented. Different MBT tools utilize the IOCO relation. IOCO is a conformance relation for LTSs [8, 9]. It is used to compare implementations to their specification. Specifically IOCO expresses that an SUT conforms to the specifications if the SUT never produces an output that cannot be produced by the specification in the same situation[10]. It is important that the SUT is input enabled LTS.

Definition 4. *For a specification s and an Implementation Under Test (IUT) i.*

$$i\ ioco\ s =_{def} \forall \sigma \in Straces(s) : out(i\ after\ \sigma) \subseteq out(s\ after\ \sigma)$$

From the definition of IOCO it follows that in any situation the possible outputs of the implementation need to be a subset of the specification. It can thus be a partial implementation or an under-specification of the actual specifications. Let us specify a tea, coffee of hot chocolate (choco) machine. Once the button is pressed the machine will dispense any of the hot beverages. The specification s is shown in figure 2.7 as an LTS in graph form.
The implementation i_0 is IOCO conformant to the specification as for every $\sigma \in$

$Straces(s)$ it holds that $out(i_0 \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$.

$$
\begin{aligned}
out(i_0 \text{ after } \epsilon) = \{\delta\} &\subseteq \{\delta\} = out(s \text{ after } \epsilon) \\
out(i_0 \text{ after } ?but) = \{!tea\} &\subseteq \{!choco, !tea, !coffee\} = out(s \text{ after } ?but) \\
out(i_0 \text{ after } ?but{\cdot}!choco) = \emptyset &\subseteq \{\delta\} = out(s \text{ after } ?but{\cdot}!choco) \\
out(i_0 \text{ after } ?but{\cdot}!tea) = \{\delta\} &\subseteq \{\delta\} = out(s \text{ after } ?but{\cdot}!tea) \\
out(i_0 \text{ after } ?but{\cdot}!coffee) = \emptyset &\subseteq \{\delta\} = out(s \text{ after } ?but{\cdot}!coffee)
\end{aligned}
$$

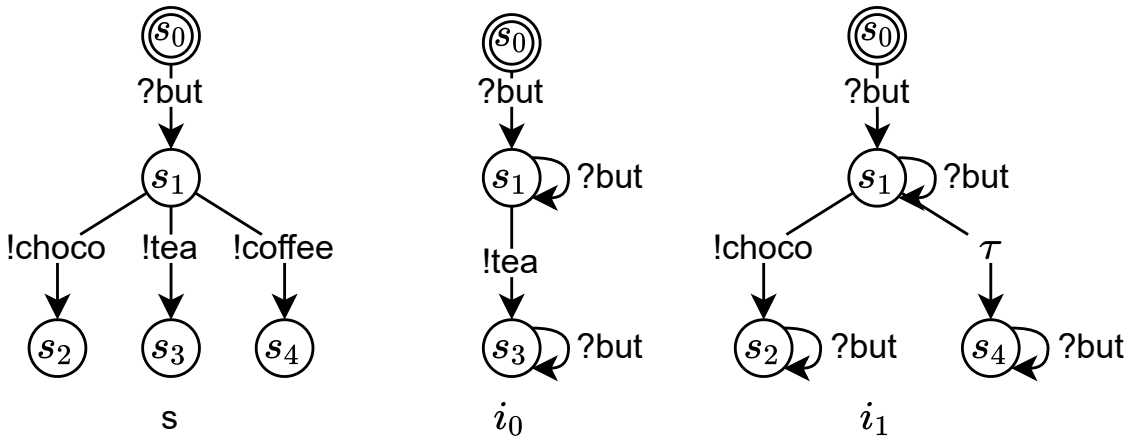

Figure 2.7: Example hot beverage machine specification and implementation LTSs

However, implementation i_1 is not IOCO conformant. Take for example the suspension trace $?but$, for the implementation $out(i_1 \text{ after ?but}) = \{!\text{choco}, \delta\}$ which is not a subset of $out(s \text{ after ?but}) = \{!\text{choco, !tea, !coffee}\}$.

2.5 Model based testing

In the previous section IOCO was introduced. IOCO is used in several MBT tools like TorX [11], JtorX [12], TorXakis [10] and Axini [13]. This section, explains how MBT works and how it uses the IOCO theory.
MBT is a form of black box testing that checks the compliance of a system to the provided model. Where white box testing looks at the internal workings, black box testing only considers external observable actions of the SUT in combination with input actions called stimuli. For a calculator performing only addition, the input is two numbers. The observable output should be the numbers added up. The model represents the expected behaviour of the SUT. A systems behaviour is the sequence of observable actions. The model describes in which order the stimuli and output can be performed and observed respectively. Such a model is manually created from

specifications, requirements, and domain knowledge. In the end the MBT tool emits a verdict, which can either be pass or fail. This pass or fail is determined based on the SUT and the model. However, the relation between the SUT and model can vary. In the previous chapter such a relation (IOCO) was discussed. This relation is used to determine if the observed actions are in accordance with the model.
It is almost never the case that the MBT tool is able to interpret the observable actions or provide the SUT with correctly formatted input. To achieve this it is common practice to wrap the SUT with a test harness that provides a translation service to and from the SUT. This can be seen in both figure 2.8 and 2.9. The blue lines are interfaces where the SUT and test harness communicate.
The model can be turned into tests in two ways. The first and most simple is offline tests, several test cases are derived from the model. These test cases are traces through the model. This means that for a pregenerated test case the non-deterministic parts have to be removed. An overview of offline tools can be seen in figure 2.8.
On the other hand, online, also known as on-the-fly tests, do not have a predetermined set of tests. The model is stepped through on-the-fly, branch decisions and checks are performed as the test progresses. Tools that use on-the-fly testing can often deal with non-determinism. They maintain the several possibilities until observed actions can resolve the non-determinism. An overview of on-the-fly tools can be seen in figure 2.9. On-the-fly tools combine the test generation and execution. Neither approaches are required to have one interface between the tool and the test harness, nor three between the test harness and SUT.

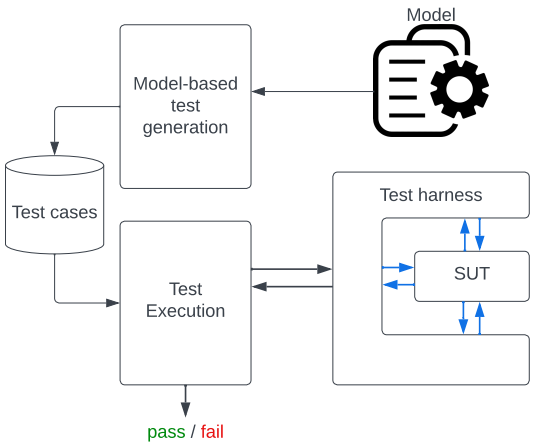

Figure 2.8: Generic diagram of an offline model-based testing tool.

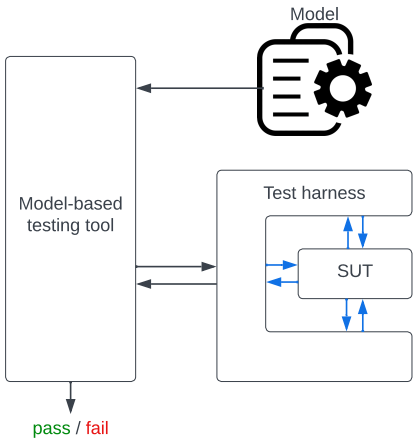

Figure 2.9: Generic diagram of an on-the-fly model-based testing tool.

This thesis utilizes the MBT tools TorXakis and Axini. Both of these tools perform tests on-the-fly. However, as mentioned earlier there are differences in approach, feature set and syntax. For example, TorXakis is incapable modeling time constraints.

Running a single test using an MBT tool does not guarantee full conformance. If the test results in a pass, for the IOCO theory it means that the used suspension trace produces expected outputs. Models that contain loops have an infinite number of possible suspension traces. Not all of these can be checked. Tools can provide strategies like transition coverage or a random approach. Test engineers should determine how long a test run should be based on the systems complexity and desired confidence in the system.

2.6 TorXakis

The notion of MBT was introduced in the previous section. TorXakis is one of the MBT tools used in this thesis. TorXakis is open source and developed at TNO[1] and the Radboud University[2].
The syntax of TorXakis will be explained. In particular, this section will go over defining new data types, functions, channels for internal and external communication, model building blocks like processes and operators, model definitions, and connecting TorXakis to a test harness using a connect definition.
TorXakis is build using the Haskell programming language[3]. This is a functional programming language. TorXakis inherits some key characteristics from Haskell. This will become apparent in the construction of new data types and functions.

Data type definition

The TorXakis syntax allows for Algebraic Data Types (ADTs) to be defined and to be use during testing. Data types can be defined in a TYPEDEF block. ADTs are recursively defined sum and product types. The simplest sum type that is commonly used in Haskell is the `Bool` type `data Bool = False | True`. This denotes that a `Bool` value can either be `False` or `True` but not both. Furthermore, a product type is a type that combines multiple other types into one. An example of a product types is a tuple. In TorXakis types can be defined as shown in listing 2.1. Here the attributes name and age are what constitutes a person. Retrieval of these attributes can be done by applying functions corresponding to the names of the attributes. For example let p = Person("Peter",24) then name(p) == "Peter" and age(p) == 24.

```
1 TYPEDEF Person ::=
2     Person{
3         name :: String;
4         age :: Int
5     }
```

[1] tno.nl
[2] ru.nl
[3] haskell.org

```
6 ENDDEF
```

Listing 2.1: Example type definition

Function definition

Functions can be defined and used like most other programming languages. However, due to TorXakis being build on top of Haskell, TorXakis also uses a functional programming approach. The control flow in TorXakis is done using only IF statements, which is simpler compared to Haskell's additional pattern matching and monads. The function definition that computes if a person is an adult (being 18+) can be seen in listing 2.2.

```
1 FUNCDEF isAdult ( p :: Person ) :: Bool ::=
2     age(p) >= 18
3 ENDDEF
```

Listing 2.2: Example function definition

Channel definition

Channels are used to perform communication either internally (i.e. communication between different processes) or externally (i.e. communication with the SUT). A channel can be defined using the CHANDEF keyword, requiring the name of the channel and the type of the communicated objects. If a channel has type Int no other types can be communicated over that specific channel. Furthermore, channels are arguments of a process and can thus undergo renaming, meaning that channels are not defined on a global scale but are used by passing them around as arguments in a local context. The only exception to this is when channels are used in the MODELDEF block that will be discussed later.

```
1 CHANDEF Channels ::=
2         calculator_input ,
3         calculator_output :: Int
4 ENDDEF
```

Listing 2.3: Example channel definition

Operators

The build-in operators in TorXakis allow for the modeling of simple concepts. Two of the most common are the sequential operator and the choice operator. The sequential operator `A >-> B` means that after performing the communication A, behaviour B is expected. The choice operator `A ## B` allows for indicating two possible options. Communication and transitions can be combined with guards to apply certain requirements.

A guard is defined as `[[expr]]` where `expr` is a boolean expression. Execution of a process or communication can be made conditional using a guard `[[ expr ]] =>> A`. Here A is only performed when `expr` evaluates to true. To perform communication over two or more channels simultaneously the `|` can be used `ChanA ? x | ChanB ? y`. The `ChanA ? x` means that some value is communicated over `ChanA` and bound to `x`.
TorXakis has special operators that allow for modeling of more complex problems. The parallel operator `A ||| B` allows one to indicate if two process run concurrently. For example `channel1 ? x ||| channel2 ? y` can lead to the following execution orders: `channel1 ? x >-> channel2 ? y`, `channel2 ? y >-> channel2 ? x` and `channel1 ? x | channel2 ? y`. When A and B are not a single communication action the interleaving becomes more complex. An example of this can be seen in figure 3.4.
Next to the parallel operator there are more ways to perform some synchronization. For example `A || B` is used to synchronize two processes over all of their channels. This means that if process A and B can not perform the exact same communication a deadlock state is reached. Two process can be synchronized over specific channels instead of all channels using `A |[Channel1, Channel2]| B`.
The disable operator `A [>> B` is used to model possible failures or unexpected issues. For example when creating or using any kind of ethernet connection in process A there is a possibility of becoming disconnected. Then B can be a process that attempts to reconnect or restart. Process A is immediately discontinued when process B starts.
Another operator is the interrupt operator `A [>< B`. It stops process A as soon as process B starts, but after process B finishes, process A is resumed. It has similar use cases as the disable operator.

Process definition

Process definitions specify the behaviour of an SUT or a part of an SUT. They can be combined to produce bigger processes using operators. A very simple thing to model would be an addition process. The process in listing 2.4 starts similar to a function. However, the arguments in between the square brackets are not variables but channels which can be used for communication. The input and output channels are meant as input and output to the SUT. Next to the brackets are parentheses, these can hold initial variables. Do note that these variables are not mutable. This means that new ones can be created but old ones can never be updated.
Furthermore, on line 2 the first input is sent to the SUT. For the SUT, input is a stimulus. Stimuli are communicated over SUT input channels. Whether a channel is an input channel or an output channel is defined in a connection definition, which will be introduced below. For convenience the 'input' channel is assumed to be an SUT input channel and 'output' is assumed to be an SUT output channel. SUT input channels can be used in two ways. The first way is by providing a concrete value using the exclamation mark. For example if the first input needs to be the Integer value 5, line 2 should be replaced with `input ! 5`. The second way is to use a question mark.

This indicates that there is no specific Integer value that needs to be communicated. Thus TorXakis will create a value and bind that to lhs. This option also allows the user to indicate constraints using guards like `input ? lhs [[lhs > 5]]`. TorXakis will attempt to resolve these guards using SAT solvers.
For channels that are output of the SUT, the exclamation mark means that the SUT's output should be exactly equal to the provided value. The question mark will bind the SUT's output to the given variable and check the optional constraint. The sequence operator `>->` is used to sent the second input used for addition. Then at last on the output channel the output of the addition is expected. This can be indicated using the exclamation mark.

```
1 PROCDEF Addition [input, output :: Int] () ::=
2         input ? lhs
3     >->
4         input ? rhs
5     >->
6         output ! lhs+rhs
7 ENDDEF
```

Listing 2.4: Example process definition

Model definition

In a model definition the process and the channels are combined. A model definition can be seen as the main function from which everything starts. The incoming and outgoing channels are defined with `CHAN {IN/OUT}` depending on if it is an SUT input or output channel. Furthermore the process that described the appropriate behaviour can be called after the `BEHAVIOUR` keyword with all its needed parameters.

```
1 MODELDEF  Main ::=
2   CHAN IN      calculator_input
3   CHAN OUT     calculator_output
4   BEHAVIOUR  Addition[calculator_input, calculator_output]()
5 ENDDEF
```

Listing 2.5: Example model definition

Connection definition

Connection definitions tie a channel to a TCP connection. These TCP connections usually connect to a test harness that sits in between TorXakis and the SUT to facilitate translation to and from TorXakis. Listing 2.6 is an example, connecting the calculator in and outputs. The `CLIENTSOCK` indicates that TorXakis will connect as a client and thus the test harness must host a server. Next to that, the `CHAN OUT` is quite contradictory to the name of the channel `calculator_input`. This is due to the fact the channel

name represents the view of the SUT while the keyword is seen from TorXakis' point of view. Furthermore, the hostname and port for the connection are indicated. Because TCP offers bi-directional communication the same connection can be used for one of the SUT's outputs.

```
1 CNECTDEF  Con ::=
2   CLIENTSOCK
3
4   CHAN OUT    calculator_input    HOST "localhost"  PORT 7890
5   ENCODE      calculator_input ? get  ->  ! toString(get)
6
7   CHAN IN     calculator_output   HOST "localhost"  PORT 7890
8   DECODE      calculator_output ! fromString(s)  <-   ? s
9 ENDDEF
```

Listing 2.6: Example connection definition

2.7 Axini

In the previous section, the MBT tool TorXakis was discussed. In this section, the workings of Axini and the syntax are introduced. Axini is the second MBT tool used in this thesis. It is produced by Axini B.V.[4] in the Netherlands. It has a proprietary platform called the Axini Modeling Platform (AMP) and engine to efficiently handle the transition system. The modeling language of Axini is build on top of the Ruby programming language[5]. Furthermore, Axini also connects to a test harness, like figure 2.9. However, everything except the SUT and harness are on the AMP in the cloud. There are some key difference between TorXakis and Axini. The first is the modeling approach. This is due to the underlying programming languages. The TorXakis language is a functional one. Axini does not inherit as much of the modeling and programming structure from its base language as TorXakis does. Nonetheless, programming in Axini is imperative. The second difference is the fact that Axini has time related capabilities. In light of this, this section will discuss: communication, functions, variables, control flow, processes and timeouts.

Communication

Axini has two types of communication channels internal and external. Internal channels are to perform communication between processes. External channels are to communicate with the SUT. This communication takes place over a websocket connected to

[4] axini.com
[5] ruby-lang.org

the AMP. The addition example can also be modeled in Axini. All the messages and channels required for the communication are shown in listing 2.7.

```
1  external 'calculator'
2
3  process('addition') {
4    timeout 5.0
5    channel('calculator') {
6      stimulus 'input', '_number' => :integer
7
8      response 'output', '_result' => :integer
9    }
10
11   receive 'input'
12   receive 'input'
13   send 'output'
14 }
```

Listing 2.7: Example Axini communication for addition

There exists a external channel on line 1 called 'calculator'. Besides that, there is one channel block in the 'addition' process. This block indicates that all messages inside it belong to the channel 'calculator'. There is thus no need to indicate the channel per message. The messages in the channel block have an identifying name and its arguments are declared. Furthermore, on line 11 to 13 the messages are send. This time the keyword 'receive' and 'send' are relative to the SUT. The SUT will receive 'input' and send 'output'. However, the values of '_number' is unknown and the output value is not checked. To prevent providing the calculator a void input, the constraint `constraint: '_number != :void'` can be applied. To keep the expected result around a variable is needed. They will be introduced next.

Variables

During the execution of a test, it can be useful to store different values. These values can be used in other parts of the model. Take for example listing 2.7, here three messages are send. However, the two input values are not stored and thus the output is never checked. This can be solved by adding an variable. Adding a variable can be done using the `var` keyword followed by a name, type and optional initial value.

```
1  process('addition') {
2    timeout 5.0
3    channel('calculator') {
4      stimulus 'input', '_number' => :integer
5
6      response 'output', '_result' => :integer
```

```
7    }
8
9    var 'expected_result', :integer, 0
10
11   receive 'input', constraint: '_number != :void', update: '
       expected_result = _number'
12   receive 'input', constraint: '_number != :void', update: '
       expected_result = expected_result + _number'
13   send 'output', constraint: '_result == expected_result'
14 }
```

Listing 2.8: The addition example updated to include constraints, a variable and variable updating.

To update the variable with the input, the '`update`' keyword can be used as shown on line 11 and 12. Here the two inputs are added up and can thus be compared to the output. The output comparison is done using a constraint on line 13.

Custom types

It is not possible to create ADTs like TorXakis. However, it is possible to create custom structs. These structs can be defined using a def macro as can be visible in listing 2.9. A new variable can be declared using the types like so: "`var 'P', person`".

```
1 def person
2   {
3     'name' => :string,
4     'age' => :integer
5   }
6 end
```

Listing 2.9: An example of a person (name and age) as custom struct type.

Next to that, one can define an enumeration of a set of concrete values.

```
1 def gender
2   Set['male', 'female', 'other']
3 end
```

Listing 2.10: An example of a enumeration/set.

A variable with this type can only be one of these concrete options. It is possible to add the type person to a set. Nevertheless, it is then not possible to have a variable be an instance of the set and a person. As that instance is not in the set.

Functions

Functions can be used to perform complex computations and manipulate variables in the model. Axini provides some functions out of the box. These functions can be applied in the model for constraining output and input values or updating local variables. However, these functions can not be used when defining custom functions. It is possible to define custom functions. The difference between the provided and custom functions is that the first can be solved as a constraint for SUT input. This means that Axini is not capable of generating any input to a custom function to satisfy a constraint. It is only capable of evaluating a custom function given its arguments. This is something that TorXakis can do.

```
1  function('equal_names', [person, person] => :boolean) { |a, b|
2    if a.name != :void
3      if b.name != :void
4        a.name == b.name
5      else
6        false
7      end
8    else
9      false
10   end
11 }
```

Listing 2.11: An example function that checks if two persons names are equal.

In the example in listing 2.11, first the presence of both names is checked. If this is the case, they can be compared, otherwise they are not equal and false is returned.

Control flow

The model in listing 2.7 only performs one addition computation on the SUT. Non-deterministic options and repeating can be modeled using the '`choice`' and '`repeat`' blocks. The '`repeat`' block can perform actions multiple times, until the keyword `stop_repetition` is encountered. The repeat block is capable of modeling non-determinism by allowing multiple options. A single non-deterministic branch can be modeled using the '`choice`' block. There also exist deterministic counterparts namely, the '`_while`' loop for repeating actions and the '`_if`' statement for a single branch. The multiple options in a repeat or choice block can be indicated with the character 'o'.

```
1  repeat {
2    o {
3      receive 'input', constraint: '_number != :void', update: '
         expected_result = _number'
4      receive 'input', constraint: '_number != :void', update: '
         expected_result = expected_result + _number'
```

```
5       send 'output', constraint: '_result == expected_result',
            update: 'expected_result = 0'
6     }
7   }
```

Listing 2.12: Repeat block with one possible branch.

Listing 2.12 shows how to encapsulate the three actions in a repeat block. The repeat does not have a branch with the `stop_repetition` keyword and will thus loop forever.

Processes

Processes combine the define messages and order of these messages to make up the (partial) behaviour of an SUT. Every action needs to be in a process. Processes can only transmit messages that are defined or imported within them. One interesting feature of Axini is it's ability to have more control over time. Within each process a timeout in seconds must be defined. This timeout will be used to wait for an observable action/reply from the SUT.
However, unlike in TorXakis, where the parallel operator can be applied on actions and processes, processes are the only level at which parallelization can be applied. An example process has already been shown in listing 2.8.

Time

In testing time can be an important factor. Waiting forever on an action that is expected to take seconds is not acceptable. As mentioned before, Axini has per process timeouts. Additionally, Axini allows for more restrictive time constraints per message. Say the output of the calculator needs to be exactly between 2 and 3 seconds. This can be modeled with the 'after' and 'before' keywords.

```
1 send 'output', constraint: '_result == expected_result',
     update: 'expected_result = 0',  after: 2, before: 3
```

The current time can be accessed using `current_time`. This can be used to apply more constraints on incoming data related to time.

Chapter 3

Testing Smart Cable Guard

In the previous chapter the background information regarding SCG, MBT and TorXakis and Axini was discussed. In this chapter, TorXakis and Axini will be applied to the SCG system. This chapter aims to provide a proof of concept implementation showing how MBT can be applied to SCG to find defects and to increase software quality. However, first the SUT and test harness will be defined. This harness is used for both TorXakis and Axini. Furthermore, the modeling of SCG in TorXakis and Axini is discussed including encountered difficulties and solutions to overcome these difficulties. Finally, the problems found within the SUT are presented. During testing both TorXakis and Axini are applied to the same firmware version, with the same setup.

3.1 System under test

The SUT is what is in the black box of MBT. The SUT has to be carefully chosen based on the systems environment and interfaces. These interfaces are the location where MBT will apply stimuli and observe actions. For SCG there are multiple possibilities. The first option is defining the SUT as one CU. This would require stubbing the SU. Since this communication is less than trivial and undocumented the effort required to achieve this would exceed the possibilities of this thesis. The second option is including the SU. This is a viable option as it limits the complexity of the test harness and does not require creating any test doubles[1]. The benefit of this is that there is no need to obtain confidence in the correctness of a test double. The third and final option is placing a whole measurement system (two CUs and two SUs) in the black box. This is another viable option as it would extend the possibilities of the model to more thoroughly check the measurement. This can only be done using two CUs as it requires information from both sides of the cable.
Due to the earlier mentioned time constraints it is only possible to create a model for one of these SUTs. The most crucial aspect is that a CU remains operational. Hence,

[1]The term test double is a generic term for a mock, fake or stub

the second option of one CU and SU will be the SUT. The test harness and SUT will utilize the following interfaces:

1. The MQTT interface, used for remote control over the CU.
2. The Arduino, it controls:
    - The power cable, this cable provides the CU with power.
    - The cable, this cable is the cable being monitored. It is used for Synchronization and Fault & PD generation.

3.2 Test harness Smart Cable Guard

The test harness connects to the SUT to make communication between the MBT tool and the SUT possible. Furthermore, SCG measures occurring cable faults and PD. These are rarely seen in the lab environment. These different inputs are thus simulated, this can be reliably done using two generators. One generator for faults and one for PD. However, these generators require physical interaction via a Transistor-Transistor Logic (TTL) input. The test harness should be able to signal these generators. To solve this issue a laptop is used to run the software side of the harness while an Arduino has been programmed to accept commands via a Serial connection. The Arduino will then trigger the TTL ports of the generators. The test harness will thus consist of a hardware layer that can interact with the environment of the SUT. This is done through the fault and PD generators as well as the power supply. Besides this, there will be software that interacts with the MQTT API. The SUT with the test harness surrounding it, is displayed in figure 3.1. Everything outside the dotted lines of the SUT box is part of the test harness.

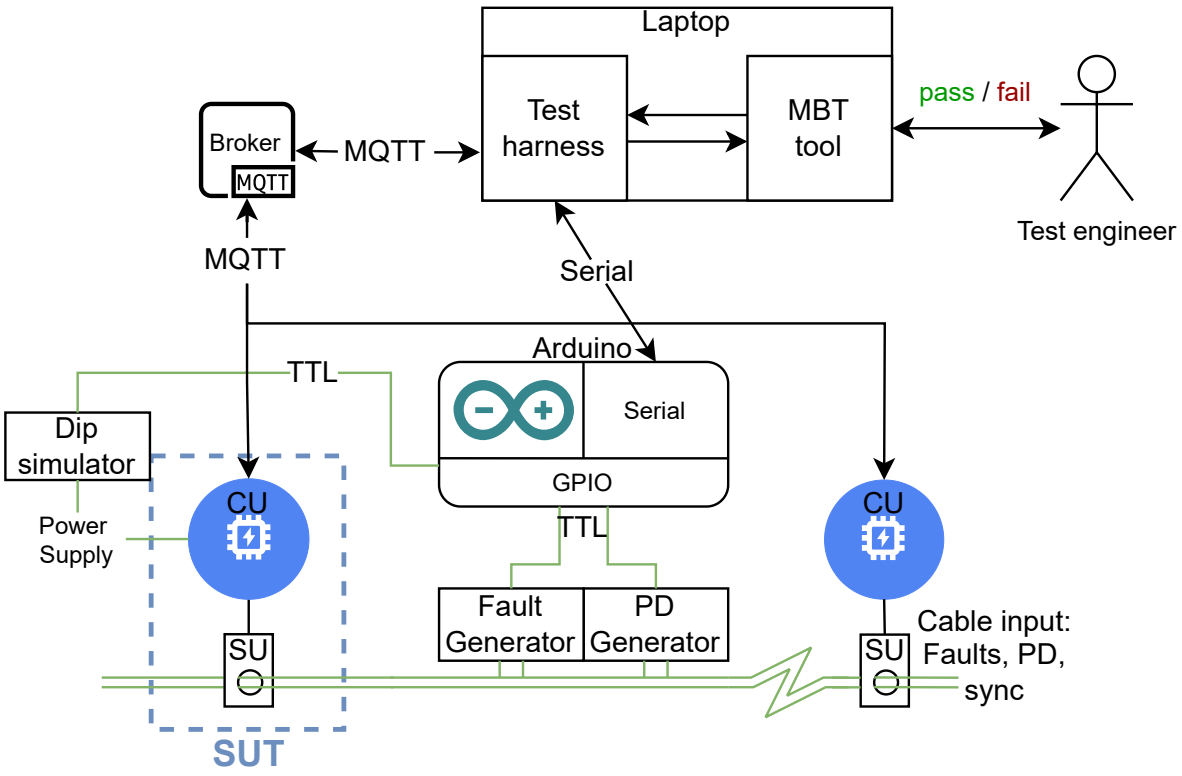

Figure 3.1: Test harness and the single CU SUT with accompanying connections

Diving deeper into the test harness there are multiple blocks. The laptop shown in figure 3.1 provides the harness with an internet connection to use the MQTT interface of the SUT and connect to outside sources. For example Axini does its decision making in the cloud, while TorXakis runs locally.
Using the General Purpose Input/Output (GPIO) pins of the Arduino three separate events can be triggered. The first is the Dip simulator. This device is used to simulate power loss on both the CU and SU of the SUT. The second is a fault generator, it injects high voltage pulses into the cable. In the real world faults can cause power loss due to the cable truly breaking or a protection mechanism kicking in. Combining the dip simulator and fault generator, it is possible to recreate the scenario where a fault causes power loss. The last device is the PD generator, which creates PD signals. As stated in figure 3.1 all three of these devices are controlled using TTL.
Faults and PD are injected into the cable. Besides that, the cable also provides another variable, namely if the system is in sync. The left and right CUs need to be in sync for the SUT to show operational behaviour. The CU on the right of figure 3.1 can cause desynchronization by adjusting several values via MQTT. This will result in different behaviour from the SUT.

The Arduino in the system can perform the TTL triggers at a rate that vastly exceeds 100Hz. This is achieved by writing directly to the register instead of using the digitalWrite function. The maximum expected frequency is 2MHz[2]. The maximum measurement frequency of faults is 100Hz. Additionally, the system should only capture a maximum of 25 pulses per minute. When monitoring the cable similarly to how SCG functions, injecting 100 fault pulses in one second looks like figure 3.2.
PD can occur at a higher frequency and thus requires faster triggering than 100Hz.

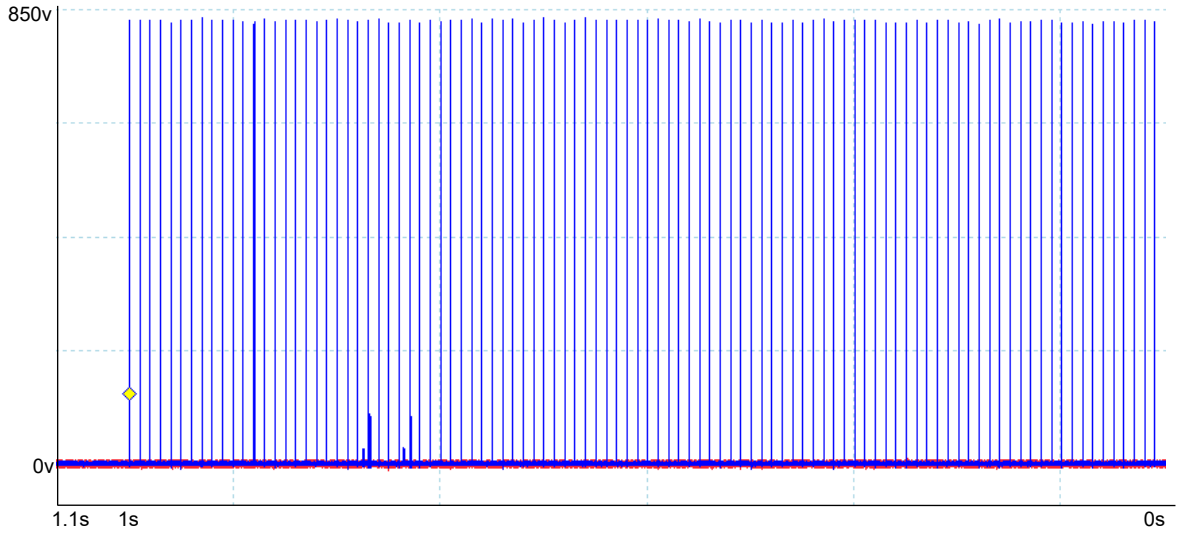

Figure 3.2: 100 Fault pulses injected in one second with normal noise.

3.3 TorXakis

Above the test harness was described including all interaction methods that are considered. In this section a closer look will be taken at model-based testing SCG using TorXakis. Firstly, the message structure and translation into TorXakis Syntax (TXS) will be discussed. After which the difficulties of time and state management that modeling SCG in TorXakis causes and how these difficulties were handled is discussed in detail.

3.3.1 Message structure

Within TorXakis all messages have been recreated as an Algebraic Data Type. Each of these messages are send over a specific TorXakis channel. The mapping of channels, messages to interfaces is shown in table 3.1. The message types mentioned are explained below.

[2] billporter.info/2010/08/18/ready-set-oscillate-the-fastest-way-to-change-arduino-pins

TorXakis		Test harness interface
MqttIn	$\xrightarrow{RootRequest}$	MQTT
MqttOut	$\xleftarrow{RootResponse}$	
PhysicalIn	$\xrightarrow{Trigger}$	Arduino

Table 3.1: Mapping of TorXakis channels, messages and test harness interfaces.

All messages that the SUT can transmit on the MQTT interface are translated into TorXakis ADTs. This is done so that TorXakis can reason about all information on this interface. The MQTT messages start at a root message and then branch out into Get, Set and Run messages. Which in turn branch out into specific sections like configurations, measurements, etc. This structure is recreated as an ADT with at the root the RootRequest and RootResponse shown in listing 3.1. The RootRequest shows that it is possible for TorXakis to send either a Get, Set or Run request to the SUT. Besides that, RootResponse includes all messages that the SUT can generate. These messages are created when a requests warrants a reply or when certain criteria are met. For example when the CU connects to the network a Connected message is send.

```
1  TYPEDEF RootRequest ::=
2      Get {get :: GetRequestRPC}
3      | Set {set :: SetRequestRPC}
4      | Run {run :: RunRequestRPC}
5  ENDDEF
6  TYPEDEF RootResponse ::=
7      GetResponse {get_response :: GetResponseRPC}
8      | StatusCode {code :: ResponseStatusCode}
9      | Connected {cu :: IdMessage}
10     | Disconnected {cu2 :: IdMessage}
11     | PushedFaultReport {fault_report :: FaultReportMessage}
12     | ImplodingMessage {text :: String}
13 ENDDEF
```

Listing 3.1: Root communication structure

The Root instances can be unfolded to individual messages. An example of this is the location information request and response shown in listing 3.2. The location information has multiple fields: city or area name, company name, station code and station name. These can be individually or collectively requested by one message through the MQTT interface.

```
1  TYPEDEF GetLocationConfig ::=
2      Location     {city_or_area_name, company_name,
3                   station_code, station_name :: Bool}
```

```
4  ENDDEF
5
6  TYPEDEF LocationConfig ::=
7      Loc {
8          city_or_area_name,
9          company_name,
10         station_code,
11         station_name :: String
12     }
13 ENDDEF
```

Listing 3.2: TorXakis representation location retrieval and location response/state message

The request is identical to its MQTT counterpart in functionality. A subset can be requested by only setting the needed fields to true. (e.g. only the city_or_area_name). Nonetheless, the response is not identical in functionality. A valid TorXakis response must contain all fields. As a consequence the response is only capable of representing a message with all fields set. To achieve an incomplete message, every field in all messages would need to be wrapped in a custom optional type, just like the Haskell `Maybe` type. This is not implemented in such a way due to a lack of time. This would also require two almost identical implementations of the current state and the messages. Besides that, TorXakis does not allow the creation of custom Eq class implementations. This forces the user to either choose the default strict equality check or create a custom top down equality function for all possibilities in the ADT. It is not possible to overwrite the Eq class of a single type. Comparing the `Maybe` types would require a long function that individually checks every field. As such, requesting a subset is possible, but receiving a subset is not possible. To deal with this, subsets are not retrieved.
Furthermore, simple messages are added to trigger the Arduino using the PhysicalIn channel. These can be seen in listing 3.3. The `PhysicalOut` channel is not used and can be ignored

```
1 TYPEDEF Trigger ::=
2   FaultTrigger {pulses0 :: Int} |
3   PulseTrigger {pulses1 :: Int} |
4   PowerTrigger {pulses2 :: Int}
5 ENDDEF
```

Listing 3.3: TorXakis messages used to trigger control the Arduino.

3.3.2 Time management

A large part of embedded systems perform actions that are related to time [14, 15]. Think of a coffee machine or an ATM. From the users perspective it is reasonable to expect that their coffee and money will not take hours to produce. A very simple coffee machine could be modeled in TorXakis as in figure 3.3. Within the IOCO theory the

phenomena of observing no output from the SUT is called quiescence as introduced in chapter 2.3. This is a fundamental concept of modeling system behaviour [16]. TorXakis is able to determine quiescence. This is done with a variable amount of time that TorXakis waits for output until determining quiescence has occurred.

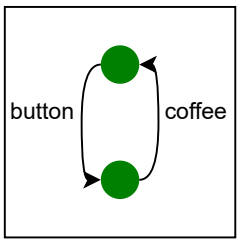

Figure 3.3: Simple coffee machine with single input button and output coffee

Specifically for quiescence to occur there must be no output of the entire SUT. When moving to more complex machines this can provide a challenge. Imagine an advanced coffee machine that is made up out of two simple coffee machines shown in figure 3.3. The advanced coffee machine is displayed in figure 3.4 and can dispense two beverages at the same time.

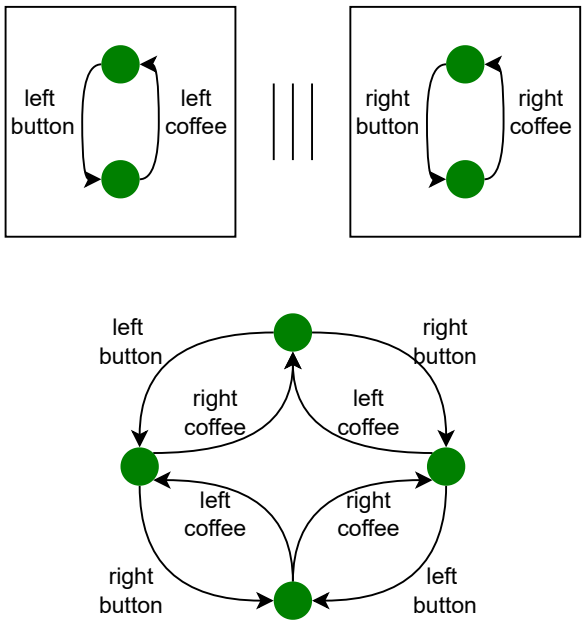

Figure 3.4: Advanced coffee machine with two input buttons and two output coffees

A coffee can be produced on the left and right simultaneously. Sadly, the left side of the SUT is broken and the coffee will never dispense. Now imagine that someone presses the left button. Quickly after attempting the left button a line of infinite people start getting coffee on the right. TorXakis will not 'see' the quiescence on the left as the

system itself is not quiescent. This is an undesirable consequence of the IOCO theory. Only if right button is not pressed sufficiently long enough will any tester using the IOCO theory eventually notice that the SUT failed due to never dispensing coffee on the left. When testing it is only possible to perform a finite number of steps within the model before needing to label the test with a PASS or FAIL. In the case of continuous interaction with the right side from the moment of the left button press, the test verdict will be a pass.

Nevertheless, there are different ways to achieve recognition of the failure. For example, synchronizing all inputs and outputs with a time channel. In this approach all messages are tagged with their corresponding time. However, caution is required. TorXakis does not provide any guarantee regarding input and output frequencies. It is thus unknown when the posted timestamp on the time channel will be consumed. There exists a real risk of inaccurate time information. With this approach it is best to let the test harness create a new timestamp on the channel when the SUT has created output or when TorXakis has created SUT input. This matches the number of timestamps to the number of messages between TorXakis and SUT and limits the time between creation and consumption of the timestamp compared to periodically adding a new timestamp. The largest drawback to this approach is the possibility of introducing false positive failures. The problem is that there is not a fixed time between the stimuli of the event and TorXakis seeing the response. If the SUT and the timestamp creation are run concurrently, it becomes possible that they intertwine in such a way that an extra message gets created. Now, the TorXakis channel holds an extra timestamp that is not expected and might fail the system if it is not dealt with.

A second approach, which is used in this thesis, moves the time management from TorXakis to the test harness. There are two reasons for this. The first being that it is simpler compared to also modeling time. The second being that this approach does not accumulate inaccuracy over time.

The least timing inaccuracy is incurred if the timer is started as close as possible to the event to be timed. As such, it is best if the SUT stimuli can simultaneously be used to start a timer in the harness, or if a starting message is send using the synchronous operator[3]. From either of these approaches the test harness starts a timer and stops it if the required result is produced on time by the SUT. If the timer is not stopped on time the test harness sends a message that is never expected by TorXakis thus always failing the test. In this thesis such a halting message is called an imploding message.

```
1 TYPEDEF Coffee ::=
2     Coffee
3     | ImplodingMessage {info :: String}
4 ENDDEF
5 PROCDEF Advanced_Coffee_Machine [ leftButton, rightButton;
    leftCoffee, rightCoffee :: Coffee ] ( ) ::=
6         Simple_Coffee_Machine [ leftButton, leftCoffee ] ( )
```

[3] torxakis.org/userdocs/stable/grammar/Synchronous_Operator.html

```
7       |||
8           Simple_Coffee_Machine [ rightButton, rightCoffee ] ( )
9   ENDDEF
10  PROCDEF Simple_Coffee_Machine [ Button, Coffee :: Coffee ] ( )
        ::=
11          Button
12      >->
13          Coffee ! Coffee
14      >->
15          Simple_Coffee_Machine [ Button, Coffee ] ( )
16  ENDDEF
```

Listing 3.4: TorXakis representation of the advanced coffee machine with imploding messages to enforce time

In listing 3.4 it is visible that the imploding message has been added. This is done so TorXakis can read the message without crashing and instead fails the test run.

```
1  class Harness:
2    def __init__():
3      ...
4
5    def torxakis_handler(self):
6      # Receive data from TorXakis
7      data = self.torxakis.recv(10240)
8      if not data:
9        break
10
11     # process TorXakis message:
12     stimuli = translate to SUT stimuli ...
13
14     if stimuli == left_button_press:
15       timer = Countdown(30, action = lambda: self.
             imploding_message("Left"))
16       self.left_timer = timer
17       timer.start()
18
19     elif stimuli == right_button_press:
20       ...idem...
21
22     self.sut.sendall(stimuli)
23
24   def sut_handler(self):
25     data = self.sut.recv(10240)
26
```

```
27      # process sut message/response
28      resp = ... translate to TorXakis ...
29
30      # if desired behaviour is seen stop timer.
31      if resp == left_coffee:
32        self.left_timer.stop()
33      elif resp == right_coffee:
34        ...idem...
35
36      self.torxakis.sendall(resp)
37
38    def imploding_message(self, side: str):
39      message = ("ImplodingMessage(" + side + " coffee was too
           late)").encode('utf-8')
40      self.torxakis.sendall(message)
41
42
43  class Countdown:
44    def __init__(self, seconds_to_wait: int, action):
45      self.seconds = seconds_to_wait
46      self.action = action
47      self._stop = False
48
49    def wait_for_action(self):
50      for i in range(self.seconds):
51        if self._stop:
52          return
53        time.sleep(1)
54      if self._stop:
55        return
56      self.action()
57
58    def stop(self):
59      self._stop = True
60
61    def start(self):
62      t = Thread(target=self.wait_for_action)
63      t.start()
```

Listing 3.5: Test harness example demonstrating imploding messages in python code

The extra TorXakis message is a small change. The main adjustment is done in the test harness. Pseudo python code of such a test harness can be seen in listing 3.5. This, code is only an example on how to implement imploding messages.

The most important changes are identifying when to start and stop a specific timer as seen on line 14-20 and 31-34. A Countdown object is created with 30 seconds to countdown and a lambda function[4] to abstract from the arguments to imploding_message.

As soon as the countdown is started a new thread is spawned. This allows other processes and communication to continue. Thus starting a countdown is non blocking. In the case of the coffee machine, the countdown is stopped when coffee is produced on the corresponding side. If the countdown has been flagged to stop it does so and the message is not sent. However, when the coffee is produced too late. The timer has run out and the test harness automatically performs the provided action. This action can be any callable object. Above, the provided action sends the imploding message. Resulting in TorXakis marking the test run as failed.

3.3.3 State management

Most models have state, meaning that their behaviour depends on previous actions. Take for example the simple coffee machine from figure 3.3. Coffee is made with grounds usually stored in a hopper on top of the machine. To represent this in a model, the amount of grounds in the machine needs to be tracked. This is done in the state of the coffee machine. Let this state be how many coffees can be made with what is in the hopper. Decreasing by one every time a coffee is made.

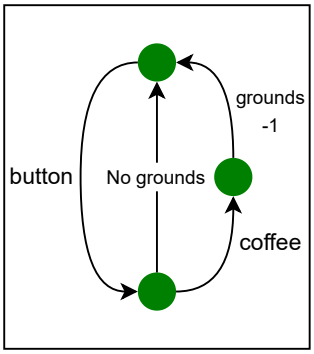

Figure 3.5: Simple coffee machine with state tracking the number of coffees that can be produced

Keeping the state within the model works well when the model is a single simple process. Because the state only requires modification by a single process at most. However, problems arise when introducing more processes that require mutable access to the shared state. Extending the hopper/grounds analogy to the advanced coffee machine with a single hopper shown in figure 3.4 results in a TorXakis process in listing 3.6.

[4] docs.python.org/3/tutorial/controlflow.html#lambdaexpressions

```
1  PROCDEF Advanced_Coffee_Machine [ leftButton, rightButton,
       leftCoffee, rightCoffee, leftError, rightError ] (grounds
       :: Int) ::=
2          Simple_Coffee_Machine [ leftButton, leftCoffee,
               leftError ] ( ? )
3       |||
4          Simple_Coffee_Machine [ rightButton, rightCoffee,
               rightError ] ( ? )
5  ENDDEF
6  PROCDEF Simple_Coffee_Machine [ Button, Coffee, Error ] (
       grounds :: Int ) ::=
7          Button
8       >->
9                [[grounds > 0]]
10               (
11                   Coffee
12               >->
13                   Simple_Coffee_Machine [ Button, Coffee, Error
                         ] ( grounds - 1 )
14               )
15         ##
16               [[grounds == 0]] =>>
17               (
18                   Error
19               >->
20                   Simple_Coffee_Machine [ Button, Coffee, Error
                         ] ( grounds )
21               )
22 ENDDEF
```

Listing 3.6: TorXakis representation of the advanced coffee machine with state

In listing 3.6 the grounds state information in the advanced coffee machine is not passed to the sub processes. As it is unclear what each simple coffee machine should get as a state. If the left and right coffee machine both get the total number of grounds then the model would be able to produce twice as many coffees as the machine. Giving each sub coffee machine half of the grounds means that one side would not be able to produce all coffees. The models used in model based testing specify the desired behaviour of a SUT [6, 17]. However, the advanced coffee machine shown above would not model the desired behaviour. Staying as close as possible to the actual desired behaviour is important.
The problem here is that the two simple coffee machines are never synchronized. This could be achieved by performing communication over internal channels once either sub process has produced coffee. This way both internal processes can keep their state

updated with the actions of the other process. However, adding a third simple coffee machine to that would require adjusting all other processes to also synchronize with the third simple coffee machine. A more elegant solution is to model the state as a separate process with which hidden synchronization occurs. This process is displayed in figure 3.7. It can be seen as a sort of global variable.

```
1  PROCDEF CoffeeState [ GetGrounds :: Int; DecreaseGrounds ] (
       grounds :: Int) ::=
2      (
3          GetGrounds ! grounds
4        >->
5          CoffeeState [GetGrounds, DecreaseGrounds] (grounds)
6      )
7    ##
8      [[grounds > 0]] =>>
9        (
10           DecreaseGrounds
11         >->
12           CoffeeState [GetGrounds, DecreaseGrounds] (grounds
               - 1)
13       )
14 ENDDEF
```

Listing 3.7: TorXakis state process for advanced coffee machine

Lines 2-6 are for retrieving the internal information. In this case that is the total number of grounds left. Lines 8-13 are for decreasing the internal grounds variable. The guard on line 8 and decreasing recursive call on line 12 guarantee that the total number of decreasing transitions that can be made is equal to the amount of grounds (i.e. the grounds variable will never become negative.)

```
1  PROCDEF Simple_Coffee_Machine [ Button, Coffee, Error;
       GetGrounds :: Int; DecreaseGrounds ] () ::=
2      Button
3    >->
4        (
5          Coffee | DecreaseGrounds
6        >->
7          Simple_Coffee_Machine [ Button, Coffee, Error,
               GetGrounds, DecreaseGrounds ] ()
8        )
9      ##
10       (
11         Error | GetGrounds ? grounds [[grounds == 0]]
```

```
12          >->
13            Simple_Coffee_Machine [ Button, Coffee, Error,
                 GetGrounds, DecreaseGrounds ] ()
14          )
15  ENDDEF
```

Listing 3.8: TorXakis model of the simple coffee machine with state process

```
 1  PROCDEF  Advanced_Coffee_Machine [ leftButton, rightButton;
       leftCoffee, rightCoffee; leftError, rightError ]  (grounds
       :: Int) ::=
 2    HIDE [GetGrounds :: Int; DecreaseGrounds] IN
 3        CoffeeState[ GetGrounds, DecreaseGrounds ] (grounds)
 4      |[ GetGrounds, DecreaseGrounds ]|
 5        (
 6          Simple_Coffee_Machine [ leftButton, leftCoffee,
                leftError, GetGrounds, DecreaseGrounds ] ()
 7        |||
 8          Simple_Coffee_Machine [ rightButton, rightCoffee,
                rightError, GetGrounds, DecreaseGrounds ] ()
 9        )
10    NI
11  ENDDEF
```

Listing 3.9: TorXakis model of the advanced coffee machine with state process

In listing 3.9 the advanced coffee machine is shown as TorXakis model, which internally uses the simple coffee machine and coffee state. On line 2 new internal channels are created and hidden. On line 3-9 the coffee state process and two interleaved simple coffee machines are synchronized with each other. Adding another simple coffee machine is simple and requires no rewriting of processes. Furthermore, there are no edge cases that need to be dealt with. Coffee can only be produced if it is also possible to decrease the total amount of grounds. Besides that, if an error is produced by the system, there should be no grounds left (The only error). Mutable access is achieved while making that access relatively safe. Caution is still required as it is rather trivial to introduce classic time-of-check time-of-use (TOCTOU) problems. For example if the retrieving of the grounds from the state process on line 11 in listing 3.8 is done at the beginning of the process. Usage of the state happens some time later, in the mean time the state could have been altered multiple times.

3.3.4 Modeling the Control Unit in TorXakis

Using the main message structure, time management and state management it is possible to define the desired behaviour of a CU. In listing 3.10 the process that maintains the accurate state is shown. This state process is more complex than the one shown in

listing 3.7. However, the idea is the same. On line 2 to 4 the process can provide the internal state, which is used in constraints. On line 6-8, the state is adjusted on the basis of a set request called 'req'. Finally, on line 10 the kept state needs to be adjusted using the response from the SUT. The response can contain information regarding performed measurements that have become available for retrieval. In some cases the request is also required to make the correct state update. This is the case when it contains identifying information regarding what measurement has been requested.

```
1  PROCDEF StateHandler [ Request :: RootRequest; Response ::
       RootResponse; State :: CuState ] ( state :: CuState ) ::=
2              State ! state
3          >->
4              StateHandler [Request, Response, State ] ( state )
5      ##
6              Request ? req
7          >->
8              StateHandler [Request, Response, State] (
                   update_state_req(req, state))
9      ##
10             Response ? resp
11         >->
12             (
13                     StateHandler [Request, Response, State] (
                           update_state_resp(resp, state))
14                 ##
15                     (
16                             Request ? req
17                         >->
18                             StateHandler
19                                 [Request, Response, State]
20                                 ( update_state_resp_req(resp,
                                       req, state))
21
22                     )
23             )
24 ENDDEF
```

Listing 3.10: CU state handler process

```
1  PROCDEF CU [ PhysicalIn :: Trigger; PhysicalOut :: Int; MqttIn
       :: RootRequest; MqttOut :: RootResponse ]  ( state ::
       CuState; cuRestarts, networkRestarts :: Int )::=
2    HIDE [ Request :: RootRequest; Response :: RootResponse;
         State :: CuState ] IN
3
```

```
4         requests_and_triggers_loop[
5           PhysicalIn, PhysicalOut,
6           MqttIn, MqttOut,
7           Request, Response, State
8         ] (cuRestarts, networkRestarts)
9
10      |[Request, Response, State]|
11
12        StateHandler
13          [Request, Response, State]
14          (state)
15    NI
16  ENDDEF
```

Listing 3.11: First part of the process defining the behaviour of a CU

The process shown in listing 3.11 is the main process of the behaviour of the CU. The multiple input and output channels connect to the physical side and MQTT interface of the SUT. It is directly visible that the state management discussed in chapter 3.3.3 has been applied. Furthermore, on line 4 the process that interleaves requests and triggers is called. This process is shown in 3.12.

```
1  PROCDEF requests_and_triggers_loop [
2                 PhysicalIn :: Trigger; PhysicalOut :: Int;
3                 MqttIn :: RootRequest; MqttOut :: RootResponse;
4                 Request :: RootRequest; Response :: RootResponse
                     ; State :: CuState
5               ] (cuRestarts, networkRestarts :: Int) ::=
6       State ? state
7     >->
8       (
9           (
10              CU_requests [
11                MqttIn, MqttOut,
12                Request, Response, State
13              ] ()
14            |||
15              Triggers[PhysicalIn, PhysicalOut, MqttOut,
                   Response] ()
16          )
17        [>>
18          ... system / network restart routines ...
19      )
```

```
20 ENDDEF
```

Listing 3.12: Process that interleaves MQTT requests and Physical triggers

These requests and triggers are performed within different processes due to the recursive calls required for looping and continuing after disabling routines. If the recursive calls are done needlessly the number of interrupt or disable branches increases. Tests that run for multiple hours to days this will become problematically slow. The system and network restart on line 18 are not shown due to size. To provide some detail, when a restart is requested the SUT disconnects and reconnects some time later.
On line 6 the internal state is retrieved from the state handler. The state is only used within the restart routines on line 18 to retrieve identifiers. This does not require an up-to-date state so TOCTOU issues do not occur.
Diving deeper into the physical side of the model and test harness. CUs are continuously looking for faults and PDs. These can be simulated using specific generators. Taking faults as an example, TorXakis can indicate that the fault generator should be triggered by sending a FaultTrigger message to the test harness. As is visible on line 2 of listing 3.13. Furthermore, TorXakis awaits the pushed FaultReport and then hands it off to the state handler to include the FaultReport in the state such that further requests can be based of that FaultReport.

```
1 PROCDEF Triggers [PhysicalIn :: Trigger; MqttOut ::
    RootResponse; Response :: RootResponse]
2         PhysicalIn ? trigger [ isFaultTrigger(trigger) ]]
3     >->
4         -- Potential 3 min delay here
5         MqttOut ? x [[ isPushedFaultReport(x) ]]
6     >->
7         Response ! x
8 ENDDEF
```

Listing 3.13: Simple Fault trigger modeled in TorXakis

Next to the trigger loop, a straightforward request loop is used. A snippet of the request loop can be seen in listing 3.14. In this listing, only the simple get request is shown. A get request based on the state is created, think of retrieving a configuration like location. The functions '`checkSutGetInputRoot`' makes sure that only specific requests are generated. The response to the request is checked against the state by applying the request to the known state in the function '`getFromState`'. The same is done within the other request branches (i.e complex get, set and run). The only extra thing they do extra is updating the state based on the request and response.

```
1 PROCDEF CU_requests [
2             MqttIn :: RootRequest; MqttOut :: RootResponse;
3             Request :: RootRequest; Response :: RootResponse;
                State :: CuState
```

```
4           ] () ::=
5           State ? state
6       >->
7           (
8                   (
9                       MqttIn ? req [[checkSutGetInputRoot(req,
                            state)]]
10                  >->
11                      MqttOut ? resp [[getFromState(req, resp,
                            state)]]
12                  >->
13                      recursive call
14                  )
15              ##
16                  Get request, retrieving new information (
                        alters state)
17              ##
18                  Set request (alters state)
19              ##
20                  Run request (alters state)
21          )
22  ENDDEF
```

Listing 3.14: Request loop modeled in TorXakis

3.4 Axini

Previously modeling SCG in TorXakis was discussed. In this section the same will be done for Axini. The problem of time and state management have not been encountered in Axini. Axini has time constructs built in and has variables on a process basis. These variables with some added internal communication between processes are enough to perform state management. In this section, the message structure, the model of SCG in Axini and the problem of alternative behaviours such as TorXakis' disable and interrupt will be discussed.

3.4.1 Message structure

The translation from SCG to Axini is less one on one compared to TorXakis. Within TorXakis new data types are equivalent to build in data types. However, Axini has a different approach, a message is not a type, but a label. This label can be accompanied by additional data. The Axini syntax creates a flat message structure as messages are not nested within each other.
As an example, the location information message is used again. The information can be

retrieved using the label "`get_location`", which results in a "`location`" label reply. This location message holds an attribute with the key "`_location`" and value struct "`location`". This structure holds the relevant fields. Every message is defined as a separate label.

```
1  stimuli 'get_location'
2  response 'location', '_location' => location
3
4  def location
5    {
6      '_city_area' => :string,
7      '_company' => :string,
8      '_station_code' => :string,
9      '_station_name' => :string
10   }
11 end
```

3.4.2 Time management

Axini provides functionality to retrieve the current time and apply timing constraints. This can be used to model that an observable action needs to be performed before or after a given number of seconds. TorXakis ignores the possible observable output and instead continues to provide the SUT with input. Axini usually considers observable output as more important than input. Consequently, Axini chooses to observe the output or lack thereof. This can lead to Axini waiting the entire amount of time to conclude quiescence. Nevertheless, it guarantees that SUT failure due to unspecified quiescence will not be missed.

3.4.3 State management

The problem of TorXakis and state management does not occur in Axini as it has variables that can be updated during test execution without performing a recursive call. Nonetheless, these variables are local to a process. Some communication among the measurement and MQTT process solve the mutual access problem.

3.4.4 Modeling the Control Unit in Axini

The model can be created using the defined labels. The two main interfaces are the physical interface and the MQTT interface. Both the physical and MQTT interface have one process each. The physical process focuses on the measurement side of the CU while the MQTT process focuses purely on communication. The physical process only uses the MQTT interface for receiving pushed messages related to performed measurements. These messages are not needed within the other processes and thus the

processes remain independent except for internal communication. The physical process is partially shown in listing 3.15. The listing shows the messages used by this process, the first one is the trigger message. This message is sent to the Arduino which triggers the fault generator. The field `_triggers` indicates the number of triggers per second. The SUT should detect the faults and create a report. This report is required to be published within 3 minutes (180 seconds) after the event. This can simply be indicated using the '`before`' keyword. If a report is published, the other process is informed of the `_date_time` of the report, such that it can base new requests on this report. The pulse trigger and power trigger are constructed in a similar way compared to what is shown.

```
1  process('physical') {
2    timeout 240.0
3    channel('physical'){
4      stimulus 'fault_trigger', '_triggers' => :integer
5      ...
6    }
7    channel('mqtt'){
8      response 'fault_report', '_pulses' => :integer, '
         _date_time' => :integer, '_power_supply_lost' => :
         boolean
9      ...
10   }
11   channel('internal'){
12     response 'fault_report_notification', '_date_time' => :
         integer
13     ...
14   }
15
16   var 'fault_pulses', :integer
17   var 'fault_pulse_date_time', :integer
18   ...
19
20   repeat {
21     o{
22       receive 'fault_trigger', constraint: "_triggers > 0 &&
           _triggers <= 100", update: 'fault_pulses = _triggers'
23       send 'fault_report', constraint: '_pulses == min(25,
           fault_pulses) && _power_supply_lost == false', before:
            180.0, update: 'fault_pulse_date_time = _date_time';
           update('fault_pulses = 0')
24       send 'fault_report_notification', constraint: "
           _date_time == fault_pulse_date_time"
```

```
25        }
26        ...
27    }
28  }
```

Listing 3.15: First process of SCG model in Axini

The Axini trigger process is similar to the TorXakis implementation. However, the request process implementation is quite different. The Axini process is flat, because messages are not nested like ADTs in TorXakis. In listing 3.16 the requests process is shown with the earlier talked about location and fault messages. Any other messages have been removed to fit in this paper.
The two constructs of `@config.fetch(:stub)` and `urgent: true` have not been introduced yet. The first fetches a value with the key 'stub' from the config struct called 'config'. The second tells Axini, to take the transition over others when it is available. This is useful to perform internal communication actions as soon as possible instead of other actions.
On line 40, it can be seen that the constraint on `set_location` enforces that it does not sent an empty location. Furthermore, the flat message structure can be seen. The flat structure creates more points where possible alternative behaviour should be made available. The problem will be discussed in more detail in the following section.

```
1  process('requests'){
2    timeout 120.0
3    channel('mqtt'){
4      # get commands
5      stimuli 'get_location'
6
7      # get replies
8      response 'location', '_location' => location
9
10     # response for retrieved fault reports
11     response 'fault_reports_response', '_pulses' => [:integer
         ], '_date_times' => [:integer]
12
13     # set commands
14     stimulus 'set_location', '_location' => location
15
16     # run commands
17     stimulus 'run_delete_fault_report', '_su_id' => :string, '
         _date_time' => :integer
18
19   }
20   channel ('internal'){
```

```
21      stimulus 'fault_report_notification', '_date_time' => :
            integer
22    }
23
24    var 'pushed_fault_date_times', [:integer], []
25    if !@config.fetch(:stub)
26      var 'loc', location, init_location
27    end
28    if @config.fetch(:stub)
29      var 'loc', location, init_location_stub
30    end
31
32    var 'temp_fault_report_date_time', :integer, 0
33
34    repeat{
35      o{
36        receive 'get_location'
37        send 'location', constraint: "_location == loc"
38      }
39      o{
40        receive 'set_location', constraint: "_location['
              _city_area'] != :void && _location['_company'] != :
              void && _location['_station_code'] != :void &&
              _location['_station_name'] != :void", update: "loc =
              _location"
41        send 'status_code', constraint: "_code == 200"
42      }
43      o{
44        send 'fault_report_notification', constraint: '
              _date_time != :void', update: "pushed_fault_date_times
               = append(pushed_fault_date_times, _date_time)",
              urgent: true
45        update("pushed_fault_date_times = limit(
              pushed_fault_date_times, 1000)")
46      }
47      o{
48        receive 'get_fault_report', constraint: "_su_id == #{
              @config.fetch(:su_id)} && _date_time in
              pushed_fault_date_times", update: "
              temp_fault_report_date_time = _date_time"
49        send 'fault_reports_response', constraint: 'length(
              _date_times) == 1 && _date_times[0] ==
              temp_fault_report_date_time'
```

```
50      }
51      o{
52        receive 'run_delete_fault_report', constraint: "_su_id
            == #{@config.fetch(:su_id)} && _date_time in
            pushed_fault_date_times", update: "
            pushed_fault_date_times = reject(
            pushed_fault_date_times, '{|e| e == _date_time}')"
53        send 'status_code', constraint: "_code == 200"
54      }
55      ...
56    }
57  }
```

Listing 3.16: Requests process in Axini

3.4.5 Alternative behaviour

Besides the main behaviour, a SUT could have alternative behaviour. An example of alternative behaviour is error handling. If it is possible to perceive the alternative behaviour, it needs to be accounted for within the model. In SCG there are several examples of alternative behaviour like rebooting, updating, connection loss, frequent repeated connection loss and the application freezing or crashing. In these scenarios, the behaviour has been specified. Take for example the reboot process, it is possible to model this procedure in a process. However, the other processes are unaware of the reboot and expect the SUT to show normal behaviour. This can result in wrongfully concluding test failure.

Notifying the other processes can be done using internal communication. However, the notification of such a problem should be available in every state. LTS lend themselves well to interleaving these extra possibilities [7, 4]. However, Axini does not support interleaving.

Recreating the TorXakis disable and interrupt constructs in Axini can be done manually for small models. A disable is a choice between continuing the original path or performing the disable. The interrupt can be modeled as the choice between the interruption or τ. The problem is the number of points where these extra options need to be added. For complex models it is undesirable to do this manually. To solve this, Axini can add the disable and interrupt constructs. An example of the possible Axini syntax of the disable block is shown in listing 3.17, this block provides the TorXakis equivalent of `A [>> B]`.

```
1  disable{
2      A
3  }
4  with{
5      B
```

```
6 }
```

Listing 3.17: Possible syntax for Axini to model the TorXakis disable construct (`A [>> B]`)

3.5 Found issues

During the testing of SCG several issues were found on the CU. TorXakis and Axini have been applied to the same software versions under the same simulated conditions. All issues were found by both TorXakis and Axini. The following issues have been found:

- Index out of bounds – The CU does not push all available measurement data. This would consume too much data to transmit. However, in some cases it is interesting to be able to analyze the raw measurements. The information can be retrieved from the CU using a pulling method. However, the user is not aware of which raw measurements are available. A list of measurements can be retrieved by indicating how many objects to retrieve. This leads to an index being out of bounds when the user requests more objects than are available.
- Dereferencing a null pointer – The communication interface that was used before this thesis allowed null pointers to indicate if an object is available. However, the new communication interface does not expect nor allow this. This results in dereferencing an null pointer.
- Incorrect firmware version retrieval – The SU runs firmware and has specific normal and fallback versions. The running, normal and fallback firmware information can be queried. The request for both these versions uses the running firmware for both normal and fallback requests. Resulting in showing the user the same firmware version twice.

3.6 Conclusion

This chapter aimed to show how MBT can be applied to SCG to find implementation defects and to increase software quality. This is done by providing implementations of MBT using two tools. Furthermore, this chapter provides solutions to the challenges of time management, state management and creating a controllable test harness for SCG. The expected behaviour of SCG depends on time. Modeling the expected behaviour is possible even if the tool does not support modeling time. This can be done by shifting the time management to the test harness. Nevertheless, a tool that supports timing constraints natively should be used. This enables test engineers at DNV to develop and maintain the model.
Alternative behaviour is difficult to model without interleaving constructs, which Axini does not have. There exists a limitation regarding the chosen SUT. The SUT consists

of one CU and SU, this causes the produced data of the other CU to not be taken into account. As a result, it is not possible to check if the combined data of both CUs is correct. Besides that, the Axini model can not perform modem and complete reboot actions. This is due to Axini not supporting alternative behaviour.
The current version of SCG has been under development since 2015. The application has already seen many bug fixes. However, additional problems have been found using MBT. The found bugs do not affect the crucial functions of SCG, instead they show that the current testing effort is inadequate.
The application of MBT has overlap with the current testing strategy. To identify and recommend appropriate integration of MBT in the testing strategy a comparison of MBT and other forms of testing is required.

Chapter 4

Model-based testing compared

This chapter provides a comparison between the current manual system testing and the model-based testing of SCG. The result of the comparison is the answer of the second research question, which is, "How does model-based and automated testing compare to each other and to the manual system testing currently applied to SCG?". SCG applies unit tests along with manual system tests. MBT will not be compared with these unit tests because they are performed on a different level and thus not comparable.
At first the current testing strategy applied to the SCG system is presented. Subsequently, the relation between model-based testing and manual system verification will be discussed. Finally a short conclusion is drawn.

4.1 Current testing strategy

The current testing strategy consists of unit tests and acceptance tests. The unit tests accompany the SCG application, which consists of several components. The most important and largest component of the CU is the SU controller. It retrieves measurements and frees memory blocks of the sensor. The component is vital to the operation of the sensor. This component has an instruction and branch test coverage of 39% measured using JaCoCo[1]. The results of JaCoCo for this component can be seen in appendix A. The test coverage for the SU controller is low. Coverage metrics are not useful indicators of the effectiveness of a test suite and cannot be used as a stopping criterion for testing [18]. However, low coverage does guarantee that large areas of the code base go untested. Untested code is a problem for software reliability. This is supported by Barnes and Hopkins, low code coverage leads to low trust in software quality [19].
To create a more complete testing strategy SCG started performing acceptance tests in addition to the unit tests. The acceptance tests aim to guarantee correct operation of the most crucial functions. These tests are performed within a micronet by a DNV test engineer. The micronet is the manual and simpler equivalent of the test harness

[1] eclemma.org/jacoco

displayed in figure 3.1. In total there are 27 distinct non-release specific acceptance tests. Most of these tests use the micronet to simulate a real world environment. The acceptance test suite is not exhaustive. It does not test all of the possible requests. There is simply not enough time to perform these checks manually.
The acceptance tests focus on testing detection and correct localization of faults and PD. Furthermore, the communication aspects that allow for remote problem solving are tested. Examples of what is tested are: up and downgrading, rebooting the system, rebooting the modem and retrieving log files. The acceptance tests are performed to prevent unrecoverable failure in the field, as this would require a physical site visit and possibly CU replacement. The acceptance tests do not cover, the endurance of the system, retrieving raw faults and raw PD, correct operations using a spotty mobile connection and a large part of the communication messages. These scenarios go uncovered due to money and time constraints.

4.2 Manual, automated and model-based testing

SCG uses manual acceptance test on a system level. These tests aim to guarantee specific functionality and prevent total functionality failure. However, it is easy to overlook subtle errors. An example of this is the SU firmware version retrieval bug discussed in chapter 3. In this section, manual, automated and model-based testing will be compared on development & maintenance, test execution, cost, edge cases and the re of current test cases.

4.2.1 Development & Maintenance

SCG provides multiple web applications including measurement data graphs. These applications are used by DNV personal to interact with the CUs and view the processed measurement data. These applications are also used during testing to perform actions and check measurement results. The manual tests do not need any additional applications to be developed. The applications do not provide direct access to the data nor is the behaviour of the CU observable. As a consequence, they are not suitable for automated or model-based testing. Instead, a test harness needs to be constructed to provide an interface for automated and model-based testing. This is additional development effort that can only be used during testing. Although, any interference from the web applications will be removed, resulting in an interface with direct access to the CU. Moreover, the development of model-based tests requires deeper knowledge of more aspects like the modeling tool, modeling language and conformance theories [20]. The developer needs to take into account how the tool operates as can be seen in chapter 3 with time management. As a result, developing a model takes more time.
The maintenance required for automated and model-based testing is also higher. The manual tests are loosely defined. Hence, changes to the system will not force manual tests to be rewritten. An example of this is the change from the DLMS protocol to

MQTT, none of the functionality of the CU changed. There is thus no need to rewrite the manual test cases. The protocol change would force the entire test harness to be rewritten.

4.2.2 Cost

Testing is the leading factor of the costs of software development [21]. It is logical to attempt to reduce this. The manual tests have high reoccurring test execution costs due to the labor required. Automated tests have a higher initial cost due to the required effort during the development phase compared to manual tests. Nevertheless, research has shown that automating manual tests can reduce total costs over time compared to manual testing [22, 23] Moreover, the cost of model-based testing is higher compared to automated testing [24]. This can be explained by the deep understanding of the application domain, input, output, formats, modeling tool and modeling language that MBT requires [20]. In the case of only one to three test cycles the investment will probably not be recouped. If the number of test cycles is between 4 and 19 the investment can be recouped, depending on the project. The application of MBT will almost certainly pay off if 20 or more test cycles are performed [24]. New releases of the SCG firmware are released once to twice a year. This would result in pay off after at least 10 years of development.

4.2.3 Test execution

The manual tests at SCG are defined in natural language. Each test comes with a list of preconditions, a list of steps to take and a list of postconditions. The test cases make use of the testers flexibility to interpret ambiguous test steps and scenarios. This flexibility is an advantage as less detailed information is required. Nonetheless, the ambiguity can also lead to a difference in test results among testers or even between test executions. Both automated and model-based tests require an unambiguous definition of all preconditions, test steps and postconditions. These definitions can either be given in the form of code or a model. These forms makes sure that the test steps are performed the same every time.
The manual and automated tests are performed sequentially, MBT performs the test cases concurrently without additional effort to perform tests concurrently. The concurrent execution of the test cases will cover possible interference of the different actions on each other. In the field the system is expected to perform its functionality concurrently. The concurrent execution simulates the real world expectations of the CU better compared performing test scenarios sequentially. Java is by default not thread safe, sharing objects and especially lists across threads can result in the `ConcurrentModificationException` error. These problems can also occur within the CU application.
The manual tests only last a short period of time, repeating them many times in different sequences is highly labor intensive. Besides that, automated tests can be made to be executed multiple times with random sequences of test cases. In contrast, a

model-based test usually does not have a specific end. It can be lengthened to check the conformance of the SUT to its specification for a longer period of time. Performing long tests is necessary to check the endurance and longevity of the system [25]. Besides that, testing for long periods helps to find intermittent problems that only show up in specific scenarios [26]. For instance, these scenarios might take place during the night or after some amount of operational time.

4.2.4 Edge cases

Domain knowledge can be used to apply tests in areas of a project containing edge cases and boundary conditions. Compared to model-based tests, manual tests give more freedom to testers, as unpredictable scenarios can be tested more easily [27]. Automated testing still provides the tester with this freedom, although the tester first has to implement the test scenario. This can take more time than performing the test manually. Moreover, model-based testing checks if the SUT meets the desired behaviour. The tester has little influence on the test execution. The TorXakis test selection is primarily random, it can be given guidance using test purposes[2]. The research from Marques et al. shows that MBT and ad hoc manual testing find relatively different bugs [27].

4.2.5 Diagnosis

MBT detects an error when there is a discrepancy between the expected and observed output. This does not mean that the cause was the previous stimuli. The cause could be a specific sequence of stimuli, a stimuli long ago or it could not even be a stimuli but the environment interacting with the SUT (e.g. mobile connections). In addition, MBT is likely to never run the same test twice. This makes reproducing the error hard, which in turn makes diagnosis a difficult process. On the contrary, the manual and automated test cases take the same steps every time and test specific functionality. In the case of an error, the failing functionality is known and possibly also at what stage it failed. Reproducible errors are straightforward to diagnose, as the tester does not have to come up with possible causes and investigate them.

4.2.6 Current test cases

The manual test cases form a strong basis for SCG, as they cover critical functionality. Both automated and model-based testing can contribute to the automation of the manual system tests. An automated approach could perform 24 out of 27 test cases. The three tests that can not be automated require physical interaction (e.g. button pressing, cable switching).
Furthermore, four of the 27 tests relate more to white-box, as they take into account internal unobservable steps. They do not lend themselves well for black-box testing.

[2] torxakis.org/userdocs/stable/grammar/PurpDefs.html

As a consequence, MBT can replicate less test cases. The SUT, model and test harness used in 3 incorporates 14 out of the 24 automatable test cases. Extending the SUT, model and test harness will increase the tests that the model can cover to 20.

4.3 Conclusion

Manual tests require less effort to develop and maintain. Development and maintenance of the model-based tests requires the most effort. Automated test sit in the middle. Furthermore, the largest cost item in software development is the testing itself. The testing costs can be reduced by automated or model-based testing given enough time to spread the high initial investment. At the current frequency of new releases within SCG recouping the MBT investment will take 10+ years.
Executing tests manually is error prone. Moreover, manual and automated tests are performed sequentially. Concurrent execution is preferred due to its similarity with the real world and the coverage between application components. In addition, the endurance of SCG is important, the devices need to run without issue for days on end. Automated tests can be applied several times, while MBT can simply run for more steps. MBT has the advantage of performing different sequences of steps that conform to the model. Hence, MBT lends itself better for endurance testing.
Manual and automated testing provides the tester with freedom over test execution. The tester has little influence on the test execution of MBT tools, which makes it difficult to steer towards edge cases.
The failure of a model-based test does not make clear which stimuli caused the error. In contrast, errors found in manual or automated tests come with a specific reproducible sequence of stimuli and responses. Consequently, uncovering the cause of the problem usually requires more thorough analysis.
Both automated and model-based testing can cover a significant portion of the manual test cases. Neither can cover the physical test cases (e.g. button pressing). MBT is not able to incorporate some white-box tests.
Without a higher release frequency or clear goal to perform endurance tests, it is inadvisable for SCG to implement MBT. Instead, implementing automated tests would be advisable based on the lower initial costs. Manual tests are only recommended when they are used a single time.
Below in table 4.1, an overview of this comparison can be found.

Comparison aspects	Manual	Automated	Model-based	Notes
Development & Maintenance	+	++	+++	
Cost	∼	∼	∼	Depends on the number of test cycles
Test execution	+	++	+++	
Edge cases	++	++	+	
Diagnosis	+	+		
Manual Test coverage	baseline	24/27	20/27	

Table 4.1: Overview of the comparison between manual, automated and model-based testing.

Chapter 5

TorXakis and Axini compared

In this chapter the used MBT tools TorXakis and Axini will be compared. This comparison will be done on the modeling languages, communication functions, error messages, documentation, visualization, conformance theory and constraint solving. An overview of these aspects for each tool will be given in the conclusion.

5.1 The modeling language

The TorXakis modeling language is based on a functional language while Axini's approach is imperative. The TorXakis language is simple and the available constructs are straight forward. However, the functional paradigm is generally considered to be difficult to master [28]. Additionally, TorXakis comes with its own quirks. For example the equality check `==` does not bind stronger than the logical AND (`/\`) operator. This results in the expression `A == True /\ B == "Hoi"`, where A is of type Bool and B is of type String, to not type correctly.
Furthermore, the TorXakis language uses symbols to construct a model while the Axini modeling language makes use of keywords instead. For example, in Axini constraints are applied to an action using `constraint: "..."` where TorXakis uses `[[...]]`. The former is more clear to those without experience in TorXakis.
Additionally, modeling in Axini is verbose compared to TorXakis. In both cases the actions need to be defined. In TorXakis an ADT can be used that represent multiple valid instances. TorXakis will select one of these instances that meet the given constraint. To model the several instances within Axini, every instance needs to be declared and added as a communication option separately.

5.2 Communication

Model-based testing revolves around observable actions and stimuli. This comes in the form of communication messages. The test harness provides the MBT tool with a translation service to and from the SUT. TorXakis communicates user defined types, which are translated into either strings or XML formatted strings. The TorXakis serializations are unique per type. As an illustration, the instance A("lorum", 1) of the type Example shown in listing 5.1 would produce the XML formatted output displayed in listing 5.2.

```
1 TYPEDEF Example ::=
2     A {message :: String; id :: Int} |
3     B {error :: String}
4 ENDDEF
```

Listing 5.1: Example TorXakis type

```
1 <TorXakisMsg>
2     <A><message>lorum</message><id>1</id></A>
3 </TorXakisMsg>
```

Listing 5.2: XML formatted output of A("lorum", 1)

Axini transforms the structures into protobuf[1] messages. Protobuf is a compact language agnostic message format.
In contrast with TorXakis, Axini's messages are created from a generic structure where the name of the field is stored separately from the value. An equivalent to the TorXakis A instance of the Example type can be seen in listing 5.3. The structure exampleA can be added to a stimulus or response. The resulting generic structure of `{'message' => "lorum",'id' => 1}` added to a label under the `_exampleA` key, can be seen in listing 5.4. The generic structure is clearly visible as key, value pairs.

```
1 def exampleA
2   {
3     'message' => :string,
4     'id' => :integer,
5   }
6 end
```

Listing 5.3: Axini structure definition

```
1 label: "Example"
2 channel: "mqtt"
3 parameters {
4   name: "_exampleA"
```

[1] developers.google.com/protocol-buffers

```
5    value {
6      struct {
7        entries {
8          key {
9            string: "message"
10         }
11         value {
12           string: "lorum"
13         }
14       }
15       entries {
16         key {
17           string: "id"
18         }
19         value {
20           integer: 1
21         }
22       }
23     }
24   }
25 }
```

Listing 5.4: Protobuf message printed as JSON

The approach of Axini has the advantage that the message itself can not be created improperly due to a strict API that creates the messages. A downside to the generic approach is that it requires traversal and construction of the generic structure when translating messages. Besides that, the data inserted into the message can still be wrong in both approaches. Besides that, the data in the message can not be type checked. A more type safe approach would be to turn the messages into non-generic protobuf messages. The protobuf type definitions can automatically be turned into a library. The generated library ensures that the correct types are used in the message. This approach is more applicable to TorXakis as it uses strict typing. The Axini structures can be used interchangeably if the field and type combinations are the same. This would be difficult to represent in a non-generic protobuf message.
The TorXakis ADT constructs can be mapped to protobuf. A product type can be translated into a specific protobuf message. For example the A instance of the Example type can be translated to:

```
1 message A{
2     string message = 1;
3     int64 id = 2;
4 }
```

A sum type can be translated into a 'oneof' of the translation of the different instances. For example the Example type can be translated to:

```
1 message Example {
2     oneof example_oneof {
3         A a = 1;
4         B b = 2;
5     }
6 }
```

This will have the same behaviour as the sum type. However, there is one catch, TorXakis' fields are required, while protobuf's fields are not required. Fields that are not set will provide the default value. To know if a field was set or not all fields can be marked as 'optional' to be able to check presence. Protobuf is not the only possible message format, other possibilities are Cap'n Proto[2] and Thrift[3].

5.3 Functions

In both tools, functions can be used to manipulate data structures. Axini comes with several build-in functions, including generic functions applicable to lists. Some of these functions are only accessible in the model and not in user defined function. On the other hand, TorXakis only comes with basic integer, boolean, string operators and function to retrieve attributes. A generic function such as 'contains' requires manual creation for each type. Furthermore, TorXakis type checks user defined functions to the best of its ability. Besides that, TorXakis allows for the evaluation of a function, which can be used to test functions. Axini provides no static checks on user defined functions and no way to test them.

5.4 Error messages

The compiler provides a developer with feedback. In the case of a problem with the code an error message is provided. Error messages play an important role in debugging and learning to program [29]. Error messages steer the developer into the direction of the problem, attempting to provide them with the information required to find and solve the problem. The error messages in TorXakis leave much to be desired. TorXakis often presents the user with a long and difficult to understand error message. In listing 5.5 it can be seen that there are two Error instance that both have an attribute called message.

```
1 TYPEDEF Error ::=
2     ErrorA {message :: String} |
```

[2] capnproto.org
[3] thrift.apache.org/docs/idl

```
3       ErrorB {message :: String}
4 ENDDEF
```

Listing 5.5: TorXakis type that produces an error without location.

This example code is erroneous as the auto generated functions to retrieve the message attribute, will have the same name and type. This produces a double defined function error as shown in listing 5.6.

```
1 MultipleDefinitions Function at <no location>: Functions
    double defined: [FuncId {name = "message", unid = 0,
    funcargs = [SortId {name = "Error", unid = 0}], funcsort =
    SortId {name = "String", unid = 0}}]
```

Listing 5.6: Error message produced from TorXakis code shown in listing 5.5.

In this listing it can be seen that the error message includes cryptic information and the location of "no location". The cryptic information is the function signature. It looks unrecognizable because the internal representation is printed, instead of how it is represented from a users point of view. This makes it harder for the user to relate the error to their TorXakis code.
The function is generated, as a consequence it is logical that "no location" is provided. However, it would be more useful to point the user into the direction of the code that results in the generation (i.e. line number 3 and column 12). Additionally, TorXakis should provide error messages that are tailored to unique causes. The provided error message in 5.6 could for example mention that it is a generated function or even that two attributes have the same name, which is the root cause of the error. This helps reduce possible confusion as the user has not defined the problematic functions. Axini's error reporting is better compared to TorXakis. In almost every case the line with the problem is highlighted and an informative error message is shown.

5.5 Documentation

Poor documentation causes many software defects and reduces the efficiency of software development [30]. The documentation is key in understanding and being able to use a modeling language. Both TorXakis and Axini provide some form of documentation. The TorXakis documentation lacks detailed information regarding some building blocks (e.g. The interrupt operator page is blank[4]). In comparison, Axini does provide this detailed information about its building blocks with small examples. Nevertheless, the TorXakis documentation does include 5 starter examples and 5 advanced examples. These examples contain explanations along with the TorXakis code. They help a be-

[4] torxakis.org/userdocs/stable/grammar/Interrupt_Operator.html

ginning user with inspiration for tackling their own modeling problems. Axini does not provide advanced examples. However, they do provide a best practice guideline.

5.6 Visualization

The behaviour of a system can be unexpected due to many reasons. For example, the developer can have wrong expectations of used code, the developer introduces an error themselves[31] or the compiler can perform optimizations incorrectly [32]. These mistakes can also happen during the development of a model. Visual representation can aid in the discovery of unexpected interpretations or consequences of the source model. Axini can transform the source model into a visual representation in the form of an LTS. TorXakis is not capable of transforming the source model into a visual representation. However, it is possible to transform a graphical representation of a state-transition system in yED[5] to a TorXakis model.

5.7 Conformance theory

In the preliminaries, the conformance theory IOCO for LTSs was introduced. Neither TorXakis or Axini use LTSs, instead they use Symbolic Transition Systems (STSs) to represent the models. STSs are transition systems that include a notion of data and control flow based on data. TorXakis uses a special IOCO variant called Symbolic Input Output Conformance (SIOCO) which is defined in [33]. SIOCO lifts the IOCO theory to symbolic transition systems. However, it does not alter the IOCO specifications [6]. Ergo, the conformance in TorXakis is precisely defined by the IOCO theory. On the other hand, Axini conformance relation is based on the IOCO theory, but does not strictly implement it. The IOCO theory does not specify time related conformance, which Axini is capable of.
Furthermore, TorXakis can only drive a single process at a time. The parallel operator can combined multiple processes into one. However, this results in possible undetectable quiescence as discussed in chapter 3. Axini chooses to wait for the observable action and thus stop driving the other process. This is can be seen when running the example in listing 5.7, where Axini does not produce any Example1 stimuli for the timeout of 1 hour (3600 seconds).

```
1 external 'A'
2
3 process('p1') {
4   timeout 2
5   channel('A') {
6     stimulus 'Example1'
7   }
```

[5] yworks.com/products/yed

```
 8
 9    repeat{
10      o{
11        receive 'Example1'
12      }
13    }
14  }
15  process('p2') {
16    timeout 3600
17    channel('A') {
18      response 'Example2'
19      stimulus 'test2'
20    }
21    repeat{
22      o{
23        send 'Example2', after: 2, before: 3600
24      }
25    }
26  }
```

Listing 5.7: Axini model example showing how Axini deals with quiescence.

5.8 Constraint satisfaction problems

The usage of constraints within the model is useful for the generation of non trivial input for the SUT. Additionally, it can be used for complex control flow through the model. Solving these constraints can be time consuming due to their complexity. This is not ideal for on-the-fly testing tools that take time into account. The computations for test decisions must not hinder test execution. TorXakis uses the Satisfiability Modulo Theories (SMT) solvers z3 or cvc4 to solve constraints during testing. On the contrary, Axini solves constraints using prolog. Research has been conducted to apply SMT solvers in Axini. The z3 implementation is not as fast as the prolog solver, but allows for solving regular expressions and constraints over user defined data types [34].
TorXakis' syntax is minimal. However, this minimal syntax provides TorXakis with the ability to solve user defined functions in constraints. Axini, in contrast, supports a limited set of functions within constraints.

5.9 Conclusion

TorXakis and Axini use different modeling styles. The languages of both tools are simple and expressive. However, Axini's language is more verbose and TorXakis' language is more difficult without experience with the functional paradigm.

Both tools require string comparison for translating messages. Besides that, Axini's generic structure requires more template code but prevents errors in the structure and serialization of the message. This is something that TorXakis' approach is incapable of, due to being entirely string based. A more strongly typed alternative would solve this problem for TorXakis.
TorXakis comes with almost no standard functions. This requires more work writing basic functions. On the other hand, Axini does provide standard functions but does not provide the user with compile time feedback regarding user defined functions.
The error messages in TorXakis are a problem that requires addressing to increase usability. They should be concise and include a best effort location instead of "no location". Furthermore, TorXakis should not provide the internal representation of functions and types. The internal representation confuses the user, instead TorXakis should refer to the original code. Axini's error messages are sufficient in steering the user into the right direction and provide useful information to solve the problem.
The documentation for both Axini and TorXakis can use more information. TorXakis can create consistent and detailed explanations about their operators. Axini can provide and explaining complex examples.
Visualization of the model can help during debugging. Only Axini has the ability to visualize the model and step through the graph. TorXakis can transform graphs to models but not the other way around.
Both tools are based on the IOCO theory. TorXakis strictly follows it, while Axini introduced extra elements that are not described in the theory. Moreover, both tools take a different approach in scenarios where an observable action and a stimuli are possible. TorXakis will perform the stimuli over waiting for the action. On the contrary, Axini will wait for the observable action.
Axini allows for modeling time constraints. As a consequence, decision making needs to be fast to not miss time constraints. TorXakis does not take time into account and can thus apply slower SMT to solve a larger variety of constraints (e.g. regular expressions). This conclusion is summarized in table 5.1.

Summary matrix	TorXakis	Axini
The modeling language	Functional	Imperative
Communication	Strings	Generic ProtoBuf
Functions	∼	∼
Error messages	∼	✓
Documentation	∼	∼
Visualization	×	✓
Conformance theory	IOCO	Based on IOCO
Constraint satisfaction problems	✓	∼

Table 5.1: A matrix providing an overview of features per tool.
✓ : Good
∼ : Requires enhancement
× : Not implemented

Chapter 6

Model-based testing in embedded systems

In this chapter the model-based testing of general embedded systems will be discussed. The model-based testing of embedded systems is different from normal software, because specific hardware needs to be taken into account. This chapter discusses the defining of the SUT, handling of the hardware in the test harness and selecting a testing tool.

6.1 System Under Test

The SUT is one of the main points of focus when model-based testing. It should be defined with care. The cost of applying MBT to an application is high and should be considered on a project by project basis [35]. For SCG one CU and one SU is taken as the SUT. This approach does not require the development of test doubles to replace parts of the system, avoiding errors in the test doubles and higher development costs. Furthermore, the modeler should consider that only observable actions can be taken into account. The SUT should have (enough) interfaces to support the desired testability and these interfaces must be reliable. Zander et al. label it as an anti-pattern when a SUT does not have sufficient reliable interfaces to test [25], which should be avoided. Nevertheless, they also note that this is not always possible for example due to a lack of time or budget. In these cases, the SUT will need to be modified or test interfaces need to be redesigned to be able to reliably observe actions and perform stimulation. The CU also contains such a problem with the LEDs, buttons and mobile network connection. The problem can be solved for the LEDs and buttons by adding signal wires to the pins on the board or in the worst case using robotics[25]. Nevertheless, mobile network connections are difficult to capture in such a way. To have any control over the mobile connection requires advanced and expensive mobile networking equipment.

6.2 Test harness

Embedded systems have a hardware layer besides the regular software. Embedded systems often rely on time and physical interactions (e.g. measurements, movements, buttons, LEDs, etc). Due to this, it is not enough to only have a software based harness. The harnesses often requires specialized equipment to achieve an observable and controllable environment. Streitferdt et al. [36] and Shin and Lim [37] applied offline MBT to embedded systems. In both cases additional hardware is used to control and measure serial in and output, time, voltages and currents. This thesis also used additional hardware. The additional hardware layer allows the testing tool to interact with the hardware interfaces off the SUT besides the normal application programming interfaces. Figure 6.1 presents an overview of MBT with the additional hardware layer in the harness.

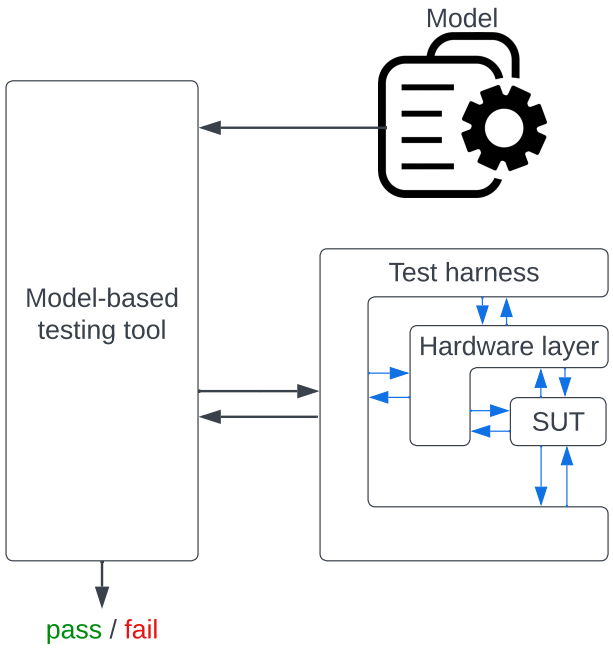

Figure 6.1: Depiction of model-based testing with the additional hardware layer.

6.3 Testing tool

There are different MBT tools, each with their own features. One of these features is online and offline testing. Online testing allows for extended test control and test reactiveness [25]. As a result more complex systems can be modeled, where multiple processes run concurrently or in parallel. Another important feature is the reasoning

of time. Not every tool can deal with time during testing. However, embedded systems often deal with time and time constraints. Applying a tool that does not natively support time to a SUT that requires timing is difficult. There exist multiple conformance relations like IOCO that can deal with time. Two examples of such a conformance relation are Timed Input Output Conformance (TIOCO) [38] and Relativized Timed Input Output Conformance (RTIOCO) [39]. The model-based testing tool called Uppaal Tron uses the latter RTIOCO relation. Additionally, online timed testing has been shown to work but timing discrepancies are unavoidable [40].
Zander et al. stipulate several other aspects of the tool that should be considered. Among these aspects are: development and maintenance effort, endurance - for longevity tests, diagnosis support, speed and time accuracy.

6.4 Conclusion

Applying MBT on a system level will limit the number of test doubles required, reducing the overall development time and costs.
Embedded systems have a hardware and a software side. To test the full system, interaction with the hardware side is required. A hardware layer is introduced into the test harness. It includes the hardware to interact with the SUT. The hardware layer has the responsibility to abstract from this hardware. This makes it straightforward to include these calls into the model. It is important that this layer can reliably perform the interactions. Complex systems are best captured by an online MBT tool to allow for more test reactiveness. Besides that, the ability to model time is an important feature when testing embedded systems. Depending on the precision of the timing requirements a tool that implements a conformance theory that takes into account time could be used.

Chapter 7

Concluding remarks

This chapter will outline the several conclusions to the research questions and provide advice based on them. Additionally, the future work will be discussed.

7.1 Conclusion

Below the conclusions per research question will be discussed. Finally advice related to the model-based testing of SCG will be given.

How can model-based testing be applied to the Smart Cable Guard system to find defects and to increase software quality?

Time management, state management and alternative behaviour are difficulties of modeling SCG. An extended test harness and a modeled state were used to successfully apply TorXakis to SCG. Axini was also successfully applied to SCG using its time modeling constructs and excluding alternative behaviour. Modeling in Axini, provided a more streamlined process usable by test engineers at DNV. Besides that, using MBT several issues have been found, indicating that the current testing strategy is insufficient.

How does model-based and automated testing of SCG compare to each other and to the manual system testing currently applied to SCG?

Manual testing requires the least amount of development effort. It also provides control over test execution, however, it is slow and labor intensive. As such, it is especially suitable for running tests few times. In addition, model-based testing requires the most amount of development effort. It also provides fast test execution, but limited control over the execution. The generation of different test step sequences is a double edged sword. It increases coverage but makes diagnosis more difficult. If enough test cycles are used the initial investment can be recouped. Moreover, automated testing provides fast and controllable test execution. It sits between manual and model-based testing in terms of development effort. The lower initial cost compared to MBT and

lower execution costs compared to manual testing result in a return on investment in fewer test cycles.

How do TorXakis and Axini compare on the aspects of their modeling language, communication functions, error messages, documentation, visualization, conformance theory and constraint solving?

Axini is the more polished tool. This can be seen in aspects such as the error messages, modeling language and visualization. Another clear difference is the choice between observing output and providing stimuli. Axini chooses to observe over stimulation, whereas TorXakis chooses stimulation over observing. Axini is capable of detecting quiescence in scenarios where TorXakis is not. Moreover, TorXakis should improve their error messages and could look into extending their modeling language to reduce repeating definitions per type or list structure.

How can the experience of model-based testing SCG be generalized to other embedded systems?

Testing an embedded systems, requires interacting with the hardware. To accomplish this a hardware layer is introduced into the test harness. The hardware layer has the responsibility to abstract from interactions on the hardware level, making it straightforward to include these interaction into the model. Furthermore, complex systems are best captured by an online MBT tool to allow more test reactiveness.

7.2 Advice

MBT can be applied to SCG effectively. In the case, that SCG wants to continuing with MBT the focus should be placed on a tool like Axini for its support, time capabilities and ease of use. However, a higher release frequency or clear goal to perform endurance tests is required. It is inadvisable for SCG to implement MBT for a one or two test cycles per year. Instead, implementing automated tests is advisable based on the lower initial costs. Besides that, the hardware still prevents all tests from being automated. There are possibilities to automate these test cases, but reliability is a serious concern. Therefore, it is recommended to keep performing these limited number of test cases manually.

7.3 Future work

The problems that the MBT tools report do not come with a line and column number. In addition, it is uncertain if the previous actions have played a role in the discovered problem. As a consequence reproducing the problem is more complicated. This leads to an interesting research avenue to see how the user can be supported by the MBT tool to trace the discovered problem back to the code of the SUT.
The generalization of applying MBT to embedded systems is based on the approach

used in this thesis and two other applications of MBT to embedded systems. The generalization could use further validation with other applications of MBT to embedded systems.
Finally, this thesis compared TorXakis and Axini with each other. A possible research avenue is the comparison of more MBT tools.

Bibliography

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE '07)*, pp. 85–103, 2007.

[2] E. M. Clarke, Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. Cyber Physical Systems Series, London, England: MIT Press, 2 ed., Dec. 2018.

[3] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-based software testing and analysis with C#*. Cambridge, England: Cambridge University Press, Nov. 2007.

[4] R. Gorrieri, *Labeled Transition Systems*, pp. 15–34. Cham: Springer International Publishing, 2017.

[5] T. Gibson-Robinson, P. Hopcroft, and R. Lazić, eds., *Concurrency, security, and puzzles*. Programming and Software Engineering, Basel, Switzerland: Springer International Publishing, 1 ed., Dec. 2016.

[6] J. Tretmans, *Model Based Testing with Labelled Transition Systems*, pp. 1–38. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[7] M. Timmer, H. Brinksma, and M. Stoelinga, *Model-Based Testing*, pp. 1–32. No. 30 in NATO Science for Peace and Security Series D: Information and Communication Security, Netherlands: IOS Press, Apr. 2011. 10.3233/978-1-60750-711-6-1.

[8] M. van der Bijl, A. Rensink, and J. Tretmans, "Compositional testing with ioco," in *Formal Approaches to Software Testing* (A. Petrenko and A. Ulrich, eds.), (Berlin, Heidelberg), pp. 86–100, Springer Berlin Heidelberg, 2004.

[9] J. Tretmans, "Testing concurrent systems: A formal approach," in *CONCUR'99 Concurrency Theory* (J. C. M. Baeten and S. Mauw, eds.), (Berlin, Heidelberg), pp. 46–65, Springer Berlin Heidelberg, 1999.

[10] G. J. Tretmans and M. Laar, "Model-based testing with torxakis: The mysteries of dropbox revisited," 2019.

[11] A. Belinfante, J. Feenstra, R. de Vries, G. Tretmans, N. Goga, L. Feijs, S. Mauw, and A. Heerink, "Formal test automation: A simple experiment," in *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems: Method and Applications* (G. Csopaki, S. Dibuz, and K. Tarnay, eds.), IFIP Conference Proceedings, (Netherlands), pp. 179–196, Kluwer Academic Publishers, 1999. null ; Conference date: 01-09-1999 Through 03-09-1999.

[12] A. Belinfante, *JTorX: exploring model-based testing*. PhD thesis, University of Twente, 2014.

[13] A. Ghaffari, "Trace coverage strategy for symbolic transition systems," 2016.

[14] P. Barry and P. Crowley, "Modern embedded computing: Designing connected, pervasive, media-rich systems," *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems*, pp. 1–518, 1 2012.

[15] S. Bhunia and M. Tehranipoor, "Hardware security: A hands-on learning approach," *Hardware Security: A Hands-on Learning Approach*, pp. 1–526, 1 2018.

[16] G. Stokkink, *Quiescent Transition Systems*. PhD thesis, Univerity of Twente, 2012.

[17] E. Roubtsova and S. Roubtsov, "A test generator for model-based testing," in *4th International Symposium on Business Modeling and Software Design (BMSD 2014) , 24-26 June, 2014 Luxembourg, Grand Duchy of Luxembourg,* (B. Shishkov, ed.), pp. 103–112, SCITEPRESS-Science and Technology Publications, Lda., 2014. conference; 4th International Symposium on Business Modeling and Software Design; 2014-06-24; 2014-06-26 ; Conference date: 24-06-2014 Through 26-06-2014.

[18] Y. Wei, B. Meyer, and M. Oriol, *Is Branch Coverage a Good Measure of Testing Effectiveness?*, pp. 194–212. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[19] D. J. Barnes and T. R. Hopkins, "Improving test coverage of LAPACK," *Appl. Algebra Engrg. Comm. Comput.*, vol. 18, pp. 209–222, May 2007.

[20] A. Neto, R. Subramanyan, M. Vieira, and G. Travassos, "A survey on model-based testing approaches: a systematic review," pp. 31–36, 01 2007.

[21] H. Reza and S. Lande, "Model based testing using software architecture," in *2010 Seventh International Conference on Information Technology: New Generations*, pp. 188–193, 2010.

[22] I. Dobles, A. Martínez, and C. Quesada-López, "Comparing the effort and effectiveness of automated and manual tests," in *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pp. 1–6, 2019.

[23] Y. Amannejad, V. Garousi, R. Irving, and Z. Sahaf, "A search-based approach for cost-effective software test automation decision support and an industrial case study," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pp. 302–311, 2014.

[24] S. Mohacsi, M. Felderer, and A. Beer, "Estimating the cost and benefit of model-based testing: A decision support procedure for the application of model-based testing in industry," pp. 382–389, 08 2015.

[25] J. Zander, I. Schieferdecker, and P. J. Mosterman, eds., *Model-based testing for embedded systems*. Computational Analysis, Synthesis, and Design of Dynamic Systems, Boca Raton, FL: CRC Press, May 2010.

[26] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," 2020.

[27] A. Marques, F. Ramalho, and W. L. Andrade, "Comparing model-based testing with traditional testing strategies: An empirical study," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pp. 264–273, 2014.

[28] A. Khanfor and Y. Yang, "An overview of practical impacts of functional programming," in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pp. 50–54, 2017.

[29] P. Denny, J. Prather, and B. A. Becker, "Error message readability and novice debugging performance," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, (New York, NY, USA), p. 480–486, Association for Computing Machinery, 2020.

[30] D. L. Parnas, *Precise Documentation: The Key to Better Software*, pp. 125–148. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[31] B. Nagaria and T. Hall, "How software developers mitigate their errors when developing code," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 1853–1867, 2022.

[32] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?," in *APSys*, 2012.

[33] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "Test generation based on symbolic specifications," in *Formal Approaches to Software Testing* (J. Grabowski and B. Nielsen, eds.), (Berlin, Heidelberg), pp. 1–15, Springer Berlin Heidelberg, 2005.

[34] F. de Geus, "On the use of smt solvers in model-based testing," 2020.

[35] S. Mohacsi, M. Felderer, and A. Beer, "Estimating the cost and benefit of model-based testing: A decision support procedure for the application of model-based testing in industry," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 382–389, IEEE, 08 2015.

[36] D. Streitferdt, F. Kantz, P. Nenninger, T. Ruschival, H. Kaul, T. Bauer, T. Hussain, and R. Eschbach, "Model-based testing of highly configurable embedded sys-

tems in the automation domain," *Int. j. embed. real-time commun. syst.*, vol. 2, pp. 22–41, Apr. 2011.

[37] K.-W. Shin and D.-J. Lim, "Model-based automatic test case generation for automotive embedded software testing," *Int. J. Automot. Technol.*, vol. 19, pp. 107–119, Feb. 2018.

[38] J. Schmaltz and J. Tretmans, "On conformance testing for timed systems," in *Formal Modeling and Analysis of Timed Systems*, pp. 250–264, 09 2008.

[39] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online testing of real-time systems using uppaal," in *Formal Approaches to Software Testing*, Lecture notes in computer science, pp. 79–94, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[40] M. Gijsen and A. Belinfante, "Timed testing with torx: The oosterschelde storm surge barrier," in *Formal Methods 2005*, 2002. Handout 8e Nederlandse Testdag; Conference date: 20-11-2002 Through 20-11-2002.

Appendix A

Code coverage Sensor Unit controller

scg20-cu-suctrl

Element		Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
	suctrl		26%		26%	333	435	892	1,253	165	236	10	24
	hal		18%		5%	85	98	191	219	67	80	6	13
	suctrl.erpc.helper		45%		30%	45	70	91	183	35	60	2	7
	analysis		78%		68%	30	86	58	228	12	39	4	12
	realtime		33%		20%	13	19	56	77	7	13	0	2
	analysis.fault		78%		75%	19	69	32	202	2	35	0	3
	logic		58%		75%	8	34	47	125	5	26	1	4
	common		72%		83%	15	59	32	104	10	38	2	4
	hal.key		0%		0%	14	14	29	29	12	12	3	3
	os		0%		0%	11	11	18	18	8	8	1	1
	hal.dbus		0%		0%	14	14	37	37	12	12	4	4
	hal.updatemanager		0%		n/a	17	17	34	34	17	17	5	5
	hal.ups		0%		0%	8	8	17	17	7	7	2	2
			0%		n/a	6	6	12	12	6	6	4	4
	converters		100%		n/a	0	16	0	24	0	16	0	3
Total		7,178 of 11,914	39%	413 of 687	39%	618	956	1,546	2,562	365	605	44	91

Figure A.1: JaCoCo code coverage report of SU controller project. The full package names have been removed.