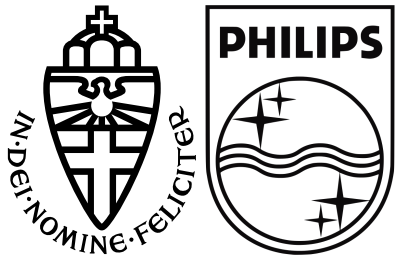
Radboud University Nijmegen

Faculty of Science



Model-Based Reliability Testing in Image-Guided Therapy Systems

Master's thesis

Internal supervisor:
dr. Jan Tretmans
tretmans@cs.ru.nl

External supervisor:
Wilma Venmans
wilma.venmans@philips.com

Second reader:
prof. dr. Marielle Stoelinga
m.i.a.stoelinga@utwente.nl

Author:
Alan Ribeiro Andrade
s1002082
alanribeiroandrade@gmail.com

August 2019

Abstract

In this thesis, we discuss reliability, software reliability, reliability testing, and model-based testing. We describe the current setup for reliability testing at Philips image-guided therapy systems. Besides that, we mention the limitations of the current approach for reliability testing using automated test scripts, such as the limited number of test cases available for that purpose. We propose the use of a model-based testing approach to improve the current situation. We devised an experiment to compare both reliability testing approaches. The results indicate significant improvements with the use of a model-based reliability testing approach. Three bugs were unveiled by the model-based approach, whereas the current reliability testing approach at Philips found no bugs. There is also an increase in the number of unique parameter values used by keywords in the model-based approach (4478) in comparison to the current setup (196). Besides that, the model-based approach executed 56 test cases, whereas the current approach only executed 16. Based on these results, we can recommend the use of a model-based reliability testing approach in case a limitation in the number of test cases for reliability testing is identified. Further research should investigate if the benefits observed in this thesis also occur in other domains.

Acknowledgments

I have received an enormous amount of help from many different people in the process of conducting this research and writing this thesis. It is finally time to write down a few words to acknowledge all the people responsible for helping me get through all the difficult times that came up on the way.

I would like to first thank my supervisor Jan Tretmans, for his feedback and for always pointing out to things I should look into better when I overlooked them. I also want to thank the invaluable feedback from Marielle Stoelinga, who helped in improving this document in a way I could not have done on my own. I am also grateful for Pierre van de Laar for always being willing to help in solving issues throughout the development of the models used in this project. I want to thank Wilma Venmans and Jacco van de Laar who found time to supervise me and give feedback during my stay at Philips. I would also like to thank Bram Jansen for finding the time to read my thesis and provide feedback on the last steps before its conclusion.

On a more personal note, I would like to thank my family back in Brazil, especially my mother Iolanda, my brother Rayllander and my sister Morgana. You gave me the motivation to keep going even though sometimes I missed home too much. We may not be together for long periods, but I love you all.

I cannot forget to thank my girlfriend Birgit, somehow we survived the long distance between Brazil and The Netherlands. There was no greater motivation for me to work as hard I could to make the best out of myself so that we could be together. For all the stressful nights that you helped me go through, thank you. I also want to thank my family-in-law, Gerard, Yvonne, and Brenda, for all the support and good times we had together.

I also want to thank all my colleagues at Philips, who were always available to answer my questions, have a coffee or lunch together. I will especially miss our lunch breaks and getting beaten up in the Philips soccer tournament with you guys.

The journey has not been easy, but it surely would have been more difficult without every single one of you.

Contents

1 Introduction

The importance of software in our lives has increased due to advances in embedded software and the Internet-of-Things (IoT). The complexity of software has also increased, and with this complexity, testing software systems have become more and more difficult.

Software testing plays an essential role in identifying problems that can be introduced in development, as well as providing evidence that requirements from the client are met. By identifying problems in an early stage, companies can avoid financial costs, for example, due to recalls. Besides that, the companies' reputation can be affected due to publicly available information on identified issues after release.

The most traditional ways of testing involve manual testing and automated test scripts. The problem with manual testing is that the tester is responsible for defining the test steps, their execution, and evaluation. Besides that, documenting and recreating the steps performed by the tester in a given test case can be difficult.

Automated test scripts bring the advantage of easy documentation and recreation of test cases, but the tester is still responsible for defining the test steps, defining what is a pass and a fail according to his understanding of how the system should behave based on requirements. Besides that, the maintenance and creation of testing scenarios can be time-consuming [12].

Other techniques aim to facilitate the testing process; one of them is model-based testing. In model-based testing, a model is created based on a specification document, which should be unambiguous and define precisely how the system should work according to predefined requirements. With such models, test cases can be automatically derived. By using such technique, a vast amount of test cases can be generated, resulting in faster and more efficient testing [13]. Assuming that the model is correct according to the specification, all test cases generated by a model-based testing tool are valid actions allowed by the system.

Reliability is the probability of observing no failures given a period of time in a system behavior. In order to measure reliability, testing is necessary. An operational profile [14] is often used to guide the system to perform tasks for a given amount of time, and its behavior is observed. The amount of time used in reliability testing can reach thousands of hours, depending on the metric or model to measure reliability.

Currently, at Philips Image-Guided Therapy (IGT) systems, there is a limited number of available test cases for reliability testing. The reason behind this problem is that at Philips IGT, within the System Integration team, a goal of 570 hours of Mean-Time-Between-Failure was defined and in order to reach that goal, testing is performed for thousands of hours. Since there are not enough test cases to cover the required amount of testing time, the available test cases run in a loop. However, software failures primarily occur when specific paths, steps, or parameters are executed. For that reason, executing the same path, steps, and parameters in a loop leaves room for improvements.

To provide the necessary amount of test cases to reach thousands of hours of testing, we propose the use of model-based testing. The use of model-based testing helps in solving this problem because by defining a model, which represents the behavior of a system under test, test cases can be generated automatically. Given the fact that the model provided for this type of testing is correct, all test cases generated in this technique are valid. In this study, we used the model-based testing tool TorXakis [1]. TorXakis can generate test cases 'on-the-fly' in a random manner, by doing so input actions can be generated according to the current state of a system and its observed output.

In reliability testing, there has been some effort in devising models [8, 10] to predict the number of failures in software systems, as well as growth models that indicate the trend for software reliability in a software development project. The literature provides guidelines [6] on how to apply such models and provide information to help in choosing

the best models for given types of applications. The term software reliability can be vague and misinterpreted, which can cause problems. In this thesis, we take a view on software reliability as a measurement, which should indicate that software behaves as it should according to specification in as many scenarios as possible. In order to verify that, our goal is to evaluate software with as many diverse testing sequences as possible. As previously mentioned, software failures occur when specific paths, steps, or parameters are executed. By testing a system with as many paths, steps, and unique parameter values as possible, we expect to identify problems, which can then be fixed and increase software reliability.

Regarding software reliability, there have also been efforts to make use of operational profiles [25, 23], which represents the user behavior. By making use of such operational profiles, testers can define test cases for a system in a realistic way, considering the behavior of users and the most used features of a system. There are challenges, however, on how to create such profiles. In case a new feature is implemented, only the expected use of a system can be defined, and unexpected behavior may be unintentionally ignored or underestimated. Another problem that can arise is that given the percentage use of features A (0.6), B (0.1) and C (0.3), the fact that feature B is only used 10% of the time, does not mean that it should not be tested thoroughly.

In this work, we focus on model-based testing, which can help in preventing unforeseen situations and achieving thorough testing of features. We aim to answer the following research questions:

- *Can we find a better way of estimating/measuring reliability than the current setup at Philips? Or in general cases?*
- *Can reliability testing at Philips benefit by the use of model-based testing? If so, how?*

In order to answer these questions, we initially performed a study on what is currently available in the field of reliability testing and prediction. Afterward, we wanted to verify the possible benefits of a model-based testing reliability approach. For that reason we designed an experiment in order to compare the current way of testing reliability at Philips and our model-based approach.

In this experiment, we submitted a Philips Azurion system to both testing approaches: execution of a reliability test case in a loop and model-based testing. Both approaches were executed for 10 hours. With the data generated from this experiment we were able to compare the number of bugs found, number of keyword executions, number of unique parameter values, similarity among test cases within testing sets, and number of test cases executed given the period of time. In order to measure the similarity within test sets, we used Jaccard(J) [7], Gower-Legendre (GL) [3], and Sokal-Sneath (SS) [18] similarity indexes. These metrics are described as the best similarity functions in the context of similarity-based selection techniques for model-based testing. This conclusion is presented in [4] based on their case studies, which evaluated 320 different similarity-based selection techniques, eight similarity functions, and ten minimizing functions.

The results indicate that an overall improvement was achieved by the use of model-based testing in reliability testing in comparison to the use of automated test scripts. These improvements include the discovery of three bugs, an increase in the diversity of test cases by the model-based approach, as well as an increase in the number of executed keywords using unique parameter values by the model-based approach (4478) in comparison to the automated test script approach (196).

In Section 2 of this thesis, we provide background information on IGT systems, reliability, and model-based testing. In Section 3, we provide an overview of the current

testing architecture at Philips IGT for reliability system integration testing, as well as an overview for the proposed architecture using model-based testing. In Section 4, we describe in-depth the reliability testing approach currently performed at Philips and our proposed approach with the use of model-based testing. In Section 5, we describe in detail the methodology used for our comparison experiment, which aims to evaluate the two approaches, including information about the evaluation metrics used in this comparison. In Section 6, we provide the results obtained by the experiment, including comments discussing the outcome of our experiment. Finally, in Section 7, we summarize the conclusions derived from this study and discuss future work that can be done based on our observations and conclusions in the topic explored in this work.

2 Background

2.1 Image-guided therapy systems

Philips produces Image-Guided Therapy (IGT) systems. These systems have as a goal the diagnosis of patients, as well as helping physicians in performing minimally invasive procedures. Philips provides a variety of IGT solutions, including CT scans, MRI, and x-ray systems. The focus of this work is in x-ray systems, specifically Azurion systems.

The x-ray technology is dated from the 19th century, but given the improvements in image processing and engineering, it is possible to obtain very high-quality images, even 3D images with low dosages of ionizing radiation (x-ray). This possibility allows physicians to perform procedures in an operation room and obtain live images of patients during procedures with low exposure to x-ray. By doing so, it is possible to evaluate more clearly possible risks and perform minimally invasive procedures with live-guidance such as endovascular procedures, where a catheter is maneuvered through arteries or veins.

In general, IGT systems are composed of a C-arc element, which has an x-ray source and a detector. The latter receives the x-ray emission and makes it possible to generate live images. This C-arc can be moved in many different axes, to allow doctors to reach the most variable positions in an operation room, where space is usually scarce. Figure 1 summarizes all possible geometry movements.

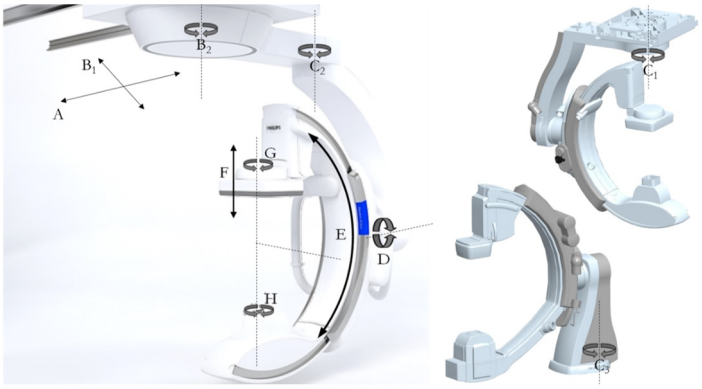


Figure 1: Geometry movements

Besides the wide variety of possible movements, there is a significant amount of features to support physicians in the workflow of diverse types of procedures, including cardiac or vascular (blood vessels) medical diagnosis and intervention, such as balloon angioplasty, where a balloon catheter is used to enlarge a blocked or narrowed blood vessel, and stent procedures, where during or right after an angioplasty, a stent (which is basically a tube) can be inserted to keep the previously blocked area unblocked. Considering the number of features gathered together in this complex system, there is a need to guarantee that all the software components, which are part of this system work in harmony together. As an example, if there is a delay in the making and displaying of an x-ray image, doctors may be visualizing the catheter in a position that is no longer live. This delay could easily cause damage to the patient's vessels and degrade its health.

On top of the complexity of the system on its own, there is a need to enforce safety, because IGT systems have a direct use on human beings. An example of a safety requirement is that the system should stop in case of collisions. In order to guarantee safety and reliability, an extensive amount of time must be invested in test planning and test execution.

This thesis will focus on reliability testing, given the importance of providing evidence that the system is reliable and tested thoroughly.

2.2 Reliability

2.2.1 What is reliability?

Reliability is often defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time [6, 5]. This definition is very general and leaves room for interpretation, which can lead to mistakes. For example, by following the above definition of reliability, there are different ways to measure the specified period of time, such as stopwatch, time scheduler, CPU time, among others. With this simple example, one can note that by using the wrong way of measuring time, when performing testing with two computers A and B, where A is faster than B, A would issue lower reliability than B, if we consider intervals of time between failures as a measure of reliability.

It is also possible that too much focus is placed on the period of time, and not enough focus is placed on the other two factors: required functions and stated conditions. The problem of focusing too much on time is that the measurement of the reliability of the system may not be realistic, because the set of required functions or stated conditions are not adequately defined or used. Naturally, with unrealistic measurements of reliability, the system might behave in an unexpected way when faced with an uncovered situation. However, reliability testing focused on time is reasonable when we consider hardware components. The wear-tear effect exists in hardware, and by observing the system behavior for long periods of time, we can estimate the capabilities of the system to behave correctly for long periods of time. That information can then be used, for example, for planned maintenance.

2.2.2 Software reliability

The rationale of reliability as a characteristic dependent on time as it is in hardware, cannot be applied to software since the behavior of software cannot degrade. An exception would be for issues related to memory management, where out-of-memory problems may appear after long periods of running time. Software failures primarily occur due to design faults, whereas hardware failures can be caused by other deficiencies in design, production, use, and maintenance [16]. Software failures usually occur when specific paths, steps, or parameters are used.

However, software reliability is often defined as the "probability of failure-free operation of a computer program in a specified environment for a specified period of time". As described in [19], most of the models for assessing software reliability make use of time. The reasoning behind the use of time comes from the assumption that the execution of a specific input in a system is random. That means that the required time for the execution of a specific sequence of inputs which can generate an unexpected output is random. Therefore, even though the failure of software is deterministic, the detection of errors is stochastic.

As explained in [15], the task of applying software-reliability engineering should be done in each phase of the development life-cycle. That means that reliability does not only start with the implementation; it should also be focused on definition, design, implementation, validation, and operation and maintenance. Therefore, in order to properly aim for software-reliability, there must be clear requirements so that designers and engineers do not make mistakes, for example, due to ambiguity in requirements.

Work in the literature also suggests that the times between failures in software tend to increase over time and failure rates in general decrease over time, once errors are identified and repaired [19]. It is important to note that even though there are studies which indicate that trend, there are several elements to be considered. One of them is how testing is performed in these studies. We can assume that by using the same set

of test cases throughout the software development of a product, the number of bugs covered by these test cases is limited. By doing so, the fact that there is a decrease in the time between failures in the software as well as the decrease in failure rate over time is only natural.

2.2.3 Reliability testing

As mentioned in [2], reliability testing is not about finding all failures in a system, and for that reason, the use of an operational profile (OP) is necessary to drive testing. An operational profile is described in [14] as "a set of disjoint (only one can occur at a time) alternatives with the probability that each will occur". This profile should represent how the system is most likely to be used by users. By creating such a profile, it is possible to perform testing representing an approximation of the real behavior of users. By using an OP, it is expected that software failures do not occur so often since the software is tested in the way it is meant to be used.

In practice, however, software testers do not always make use of an operational profile to test the software [11]. For that reason, the indicators of reliability testing may not be realistic, since they do not take into account a large variety of real user behavior.

This happens because, in development projects, there is an interest in performing as much testing as possible to guarantee that the system behaves as it should. On the other hand, there is also a high interest in releasing the product as soon as possible.

An option to facilitate the generation of test cases for testing is the use of model-based testing. This technique allows testers to create an abstraction of the system behavior and automatically generate test cases. Operational profiles correspond to the expectations on how users will use a system, but this expectation might be different from reality. For that reason, the generation of a random sequence of steps might also be beneficial in discovering bugs in behavior that is allowed but not previously considered.

Given the fact that medical systems interact directly with human beings, reliability testing plays a significant role in providing evidence that such systems function as intended over an extended period of time and that the software is reliable. In order to achieve that, reliability testing is necessary.

The purpose of reliability testing is to determine potential problems with a system design as early as possible and, to provide confidence that a system meets its reliability requirements, which are usually related to time-between failures. At Philips, the focus of reliability is to meet a Mean-Time-Between-failures of 570 hours, with a 90% confidence level.

2.3 Model-based testing

As described in [20], model-based testing consists of three main steps. First, a model, which is an abstraction of a System Under Test (SUT), should be created. This model should represent what the expected behavior of a system is. Once such a model is created, we can formally derive test cases for the SUT. These test cases are made up of sequences of interactions, which are generated based on the possible paths in our model. There are different types of test execution in model-based testing. It is possible that a set of test cases is first generated, saved, and then executed in the SUT. Another option is on-the-fly testing, where as soon as input traces are automatically generated, these are sent and executed in the SUT. In case the system provides an output, this output is observed and sent back to the testing tool. In this process, test generation and execution are performed by the testing tool, and the system behavior is verified against the formal definition of a model. One of the main benefits of on-the-fly testing is that by direct interaction with a system, it is possible to deal with non-determinism.

Differently from the traditional automated test scripts where much time is spent on creating test cases, the most time spent with model-based testing is in designing the model. In the next subsections, we provide details and examples on how to formally specify a model, as well as information details about the tool used in this thesis.

2.3.1 Labelled transition systems

Labelled transition systems are represented by a quadruple containing an initial state, a set of states, labels and transition functions.

$$(s_0, S, T, TR).$$

, where:

- $s_0 \in$ S is the initial state;
- S is a non-empty set of states;
- T is a countable set of labels;
- TR $\subseteq$ S $\times$ (T $\cup$ τ) $\times$ S, where τ is an internal transition

We can represent a simple coffee machine scheme (Figure 2) by using the structure above as follows: (s0, {s0, s1, s2}, {coffee, 50ct, button, coffee}, { <s0, button, s0>, <s0, 50ct, s1>,<s1, button ,s2>, <s1, 50ct ,s1>, <s2, coffee, s0>})

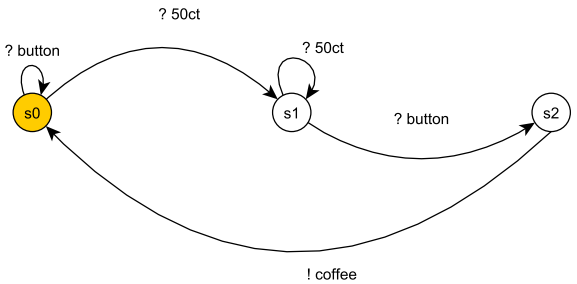


Figure 2: Labelled transition system example *lts1* (initial state: s0)

With a labelled transition system such as the one presented in Figure 2, we can derive traces, which are sequences of possible transitions. In order to illustrate traces we can use the notation of **out** and **after** [21] to represent the possible observable outputs (out) after (after) the execution of given trace. A few example of traces are denoted bellow:

$$\begin{aligned} out(lts1 \textbf{ after } ?button) &= \{\delta\} \\ out(lts1 \textbf{ after } ?button?button) &= \{\delta\} \\ out(lts1 \textbf{ after } ?button?50ct) &= \{\delta\} \\ out(lts1 \textbf{ after } ?button?50ct?button) &= \{!coffee\} \end{aligned}$$

2.3.2 Symbolic transition systems

Symbolic Transition Systems (STS) add data and data-control flow to LTS [9]. Symbolic transition systems, as defined in [9] are described as a "*tuple* $\mathcal{S} = (L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow)$ *where* L *is a set of locations,* $l_0 \in L$ *is the initial location,* $\mathcal{V}$ *is a set of location variables,* $\mathcal{I}$ *is a set of interaction variables;* $\mathcal{V} \cap \mathcal{I} = \emptyset$ *and we set* $Var = def(\mathcal{V} \cup \mathcal{I})$. Λ *is the set of gates, the constant* $\tau \notin \Lambda$ *denotes an unobservable gate;* Λ_τ *abbreviates* $\lambda \cup \{\tau\}$. *The relation* $\rightarrow \subseteq L \times \Lambda_\tau \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}} \times L$ *is the switch relation;* $l \xrightarrow{\lambda,\varphi,\rho} l'$ *abbreviates* $(l, \lambda, \varphi, \rho, l') \in \rightarrow$, *where*$\varphi$ *is the switch restriction, and* ρ *is the update mapping. We use the following functions and vocabulary:*

1. ***arity****:* $\Lambda_\tau \rightarrow \mathbb{N}$ *is the arity function*
2. ***type****(*λ*) yields a tuple of size arity(*λ*) of interaction variables for gate* λ
3. $\mathcal{S}$ *is well-defined iff arity(*τ*) = 0, type(*λ*) yields a tuple of distinct interaction variables, and* $l \xrightarrow{\lambda,\varphi,\rho} l'$ *implies free(*φ*)* $\subseteq \mathcal{V} \cup$ *type(*λ*) and* $\rho \in \mathfrak{T}(\mathcal{V} \cup type(\lambda))^{\mathcal{V}}$,
4. $\mathcal{S}(\iota)$ *is an initialized STS, where* $\iota \in \mathfrak{U}^{\mathcal{V}}$ *initializes all variables from* $\mathcal{V}$ *in* l_0.

The interpretation of an STS is defined in terms of LTS can be defined using: "Let $\mathcal{S} = (L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow)$ *be an STS. Its interpretation* $[\![\mathcal{S}]\!]_\iota$ *in the context of* $\iota \in \mathfrak{U}^{\mathcal{V}}$, *is defined as* $[\![\mathcal{S}]\!]_\iota = (L \times \mathfrak{U}^{\mathcal{V}}, (l_0, \iota), \Sigma, \rightarrow)$ *for all* $\iota \in \mathfrak{U}^{\mathcal{V}}$, *where":*

- $\Sigma = \bigcup_{\lambda \in \lambda}(\{\lambda\} \times \mathfrak{U}^{arity(\lambda)})$, *is the set of actions.*
- $\rightarrow \subseteq (L \times \mathfrak{U}^{\mathcal{V}}) \times (\Sigma \cup \{\tau\}) \times (L \times \mathfrak{U}^{\mathcal{V}})$ *is defined by the following rule:*

$$\frac{l \xrightarrow{\lambda,\varphi,\rho} l' \quad \varsigma \in \mathfrak{U}^{type(\lambda)} \quad \vartheta \in \varsigma \models \varphi \quad \vartheta' = (\vartheta \cup \varsigma)_{eval} \circ \rho}{(l, \vartheta) \xrightarrow{(\lambda, \varsigma(type(\lambda)))} (l', \vartheta')}$$

The semantics of an initialized STS $\mathcal{S}(\iota)$ *is given by the LTS* $[\![\mathcal{S}]\!]_\iota$."

The STS $(\{s_i \mid 0 \leq i \leq 2\}, s_0, \{available, counter\}, quant, \Lambda, \rightarrow)$, with Λ = ?button, ?50ct, !coffee is illustrated in Figure 3; $\rightarrow$ is given by the directed edges linking the locations. In comparison to our original coffee machine expressed using an LTS (Figure 2), with this STS, we can add data and guards to our transitions. Users can press a button to order coffee in the initial state, but nothing is observed. A user can add 50ct to the machine if there is coffee available in the machine, the counter for the current order is set to one if more 50ct coins are inserted the counter is increased. If the user presses a button, the amount of coffee given by the machine is defined by the amount of 50ct coins inserted in the machine.

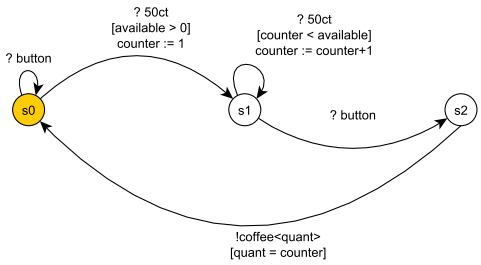


Figure 3: Symbolic transition system example (initial state: s0)

2.3.3 TorXakis

There are many different model-based testing tools with diverse ways of representing models, generating test cases, testing theories, etc. In this work, we focus on the model-based testing tool TorXakis [1]. TorXakis generates and executes test cases 'on-the-fly' in a random manner. By interacting with a SUT, TorXakis knows what input(s) and expected output(s) are available in a current state or set of states. This is different from offline testing, where a predefined sequence of actions is specified before its execution.

The following subsections describe how to define a simple illustrative model using TorXakis.

Type definition

There are predefined types in TorXakis, such as Integers, Booleans, among others. In order to develop complex models, users can define their own types. In order to define a type, the following structure is necessary:

```
TYPEDEF Keywords
    ::=
        Keyword1 {attribute1::Int; attribute2::String}
        | Keyword2 {attribute3::String; attribute4::Int}
        | Keyword3 {attribute5::Int}
ENDDEF
```

By using the above definition, we created a new type *Keywords*, which has three different constructors with different attributes.

Enumeration types can be created using type definitions, as shown with *ReturnType*, which has two possible values: *OK* and *NOK*.

```
TYPEDEF ReturnType
    ::=
        OK
        | NOK
ENDDEF
```

Personalized data types such as the ones defined above can also be used for the creation of new types, as shown below.

```
TYPEDEF SystemOutput
    ::=
        Return {rettype::ReturnType; message::String}
ENDDEF
```

Channel definition

Our tool interacts with a SUT using communication channels. We can define what is the type of data being sent over these communication channels by using the following structure:

```
CHANDEF Model ::=
    InputInterface :: Keywords;
    OutputInterface:: SystemOutput
ENDDEF
```

By using the definition above, we are specifying that our *Model* has two communication channels. An *InputInterface* that carries data of type *Keywords* and a channel *OutputInterface* of type *SystemOutput*.

Model definition

In order to create a model we need to define if the communication channels are used as input or output, as well as its behavior. The following example defines a simple model, which makes use of two channels. *InputInterface* is the input channel of our model, and *OutputInterface* is the output one. The behavior of our system in this example is expressed using a process called *execution*, which makes use of the aforementioned channels, which is described in better detail in the next paragraph.

```
MODELDEF Model ::=
    CHAN IN     InputInterface
    CHAN OUT    OutputInterface

    BEHAVIOUR
        execution [InputInterface, OutputInterface] ()
ENDDEF
```

Process definition

When defining a process, we need to specify what channels are used and if there are any parameters. In the following example, we create a process that defines the behavior of a system that receives a keyword of type *Keywords* as an input, and it is followed by an output trace of type *SystemOutput*.

The ## operator indicates that a choice is made among the possible options. In this example, only one of the three possible keywords will be executed at a time. The step operator >>> specifies that the action after the operator will only be executed if the previous actions are finished first.

```
PROCDEF execution
    [InputInterface::Keywords
    ;OutputInterface::SystemOutput] ()
    ::=

    (
    (
    InputInterface ? k1 [[
                        IF isKeyword1(k1) THEN
                            (attribute1(k1) > 0 )
                        ELSE
                            False
```

```
                        FI
                        ]]
    >-> OutputInterface ! Return(OK,"No errors")
    >-> EXIT
    )
    ##
    (
    InputInterface ? k2 [[
                        IF isKeyword2(k2) THEN
                            (attribute4(k2) == 0   )
                        ELSE
                            False
                        FI
                        ]]
    >-> OutputInterface ! Return(OK,"No errors")
    >-> EXIT
    )
    ##
    (
    InputInterface ? k3 [[
                        IF isKeyword3(k3) THEN
                            (attribute5(k3) < 0 )
                        ELSE
                            False
                        FI
                        ]]
    >-> OutputInterface ! Return(OK,"No errors")
    >-> EXIT
    )
    )
    >>> execution [InputInterface, OutputInterface] ()
ENDDEF
```

After performing an input keyword action, followed by the output observation this process goes back in a loop in such way that this sequence of steps is continuous.

With the example above, we can observe that we can add constraints for each type of action by making use of double square brackets. By doing this, we can specify a range or specific values for a given attribute. An abstraction of the simple model defined above using TorXakis can be seen in Figure 4.

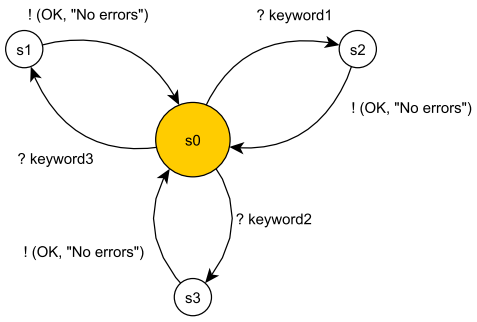


Figure 4: Simple model *spec* (initial state: s0)

2.3.4 IOCO testing theory

TorXakis makes use of the IOCO-testing theory [21], which uses labeled transition systems to represent the system models. Besides that, TorXakis makes use of an algorithm to generate test cases from a given model, which does not only create a diverse sequence of steps, but it is also able to generate input parameters for given transitions.

IOCO is an abbreviation for Input-Output COnformance. This theory describes that given an implementation i of a specification s, any experiment derived from s leads to an output from i that is expected by s. This can be summarized by the following:

$$i \textbf{ ioco } s \Leftrightarrow \forall \sigma \in traces(s) : \quad out(i\ after\ \sigma) \ \subseteq \ out(s\ after\ \sigma)$$

We can use our simple model (Figure 4) and the following simple implementation in Figure 5 to exemplify whether there is ioco conformance between them or not.

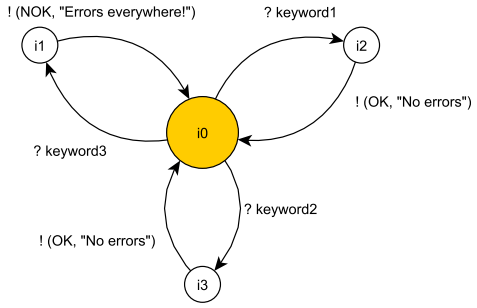


Figure 5: Simple implementation *imp* (initial state: i0)

$$\begin{aligned}
out(spec \textbf{ after } ?keyword1) &= \{!(OK, "No\ errors")\} \\
out(spec \textbf{ after } ?keyword2) &= \{!(OK, "No\ errors")\} \\
out(spec \textbf{ after } ?keyword1\ !(OK, "No\ errors")) &= \{\delta\} \\
out(spec \textbf{ after } ?keyword1\ !(OK, "No\ errors")\ ?keyword3) &= \{!(OK, "No\ errors")\}
\end{aligned}$$

$$\begin{aligned}
out(imp \textbf{ after } ?keyword1) &= \{!(OK, "No\ errors")\} \\
out(imp \textbf{ after } ?keyword2) &= \{!(OK, "No\ errors")\} \\
out(imp \textbf{ after } ?keyword1\ !(OK, "No\ errors")) &= \{\delta\} \\
out(imp \textbf{ after } ?keyword1\ !(OK, "No\ errors")\ ?keyword3) &= \{!(NOK, "Errors\ everywhere!")\}
\end{aligned}$$

Based on the definition of ioco, the *out* set of the implementation must be part of the *out* of the specification. By looking at the *out* sets after the trace *?keyword1 !(OK,"No errors") ?keyword3* we can observe that:

$$out(imp\ after\ \sigma) \ \not\subseteq \ out(spec\ after\ \sigma)$$

where σ = ?keyword1 !(OK,"No errors") ?keyword3, because

$$\{!(NOK, "Errors\ everywhere!")\} \ \not\subseteq \ \{!(OK, "No\ errors")\}$$

Test purpose

Besides the specification of a system behavior, users can also define a test purpose. The idea behind test purposes is that when testing, we do not necessarily want the behavior to be completely random, we may want to focus on specific scenarios. A simple example is the testing of a system which requires a login to get access to its full functionality. If the behavior of the system is completely random, the attempts to generate a valid username and password might take too much time. Of course, it is important to test the login functionality in this abstract scenario, but we would probably also like to spend quite some time on the rest of the functionality of the system. For that reason, test purposes can be used.

Test purposes make it possible to guide our original model to a more realistic usage of the system. A simple example of how a test purpose is defined below and depicted in Figure 6.

```
PURPDEF myPurpose ::=
    CHAN IN     InputInterface
    CHAN OUT    OutputInterface

    GOAL executeKeywordOneNTimes
        ::=
        executeKeywordOneNTimes [InputInterface,OutputInterface] (0,20)

ENDDEF

PROCDEF executeKeywordOneNTimes
    [ InputInterface :: Keywords; OutputInterface::SystemOutput] (counter, n::Int) HIT ::=

        (

            ([[counter < n]] =>>
                (InputInterface ? k1
                    [[IF isKeyword1(k1) THEN
                          (attribute1(k1) > 0 )
                      ELSE
                          False
                      FI
                    ]]
                >-> OutputInterface ! Return(OK,"No errors")
                >-> EXIT
                )
            >>> executeKeywordOneNTimes [ InputInterface, OutputInterface] (counter+1, n)
            )

        ##

            ([[counter >= n]] =>>
                (InputInterface  !  Keyword2 ("end",0)
                >-> HIT)
            )


        )


ENDDEF
```

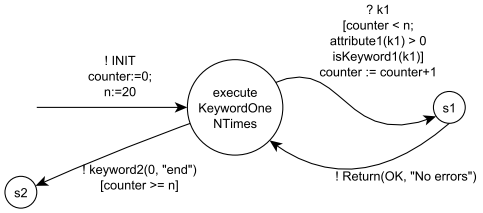


Figure 6: Simple purpose (initial state: executeKeywordOneNTimes)

With the test purpose defined above, we can set a goal of performing the execution of the keyword1 20 times, followed by the execution of keyword2 once. We could also define that a specific sequence of keywords is chosen before the beginning of the execution of our 20 steps, or that a choice among other keywords besides keyword1 and keyword2 could be made. We can add any constraint in the behavior of our test purpose, as long

as it is possible to be executed on our original model. As demonstrated in the example above, by using test purposes we can limit testing to a subset of possibilities in a given model.

3 Test architecture

3.1 Current test architecture overview

The current way of testing reliability at Philips in a system level makes use of a Test Framework, which is responsible for connecting all the components that are used for testing, generation of reports, etc. The test scripts are written using a keyword-driven approach. System-Level Keywords (SLKs) are made up of Elementary Level Keywords (ELKs) which make use of adapters that make use of interfaces to communicate with diverse components from the Suite PC, such as Imaging, Viewing, etc. The Suite PC is responsible for executing the actions sent via adapters from the Side PC into the real IGT system. SLKs provide an abstraction to test a system. In a generic and generalized example, the action of activating x-ray emission (fluoroscopy, exposure, etc.) involves several actions, such as changes in diverse signals. Each change in a signal can be seen as an ELK and an SLK such as ActivateFluoro_Start is made up of all the ELKs responsible for the changes in the signal. There is a one-to-many relationship between SLKs and ELKs, respectively. The use of keywords allows testers to have an abstraction layer on top of the direct use of adapter methods, which facilitates the development of scripts and reduces the amount of time required to devise test scripts. These test scripts are stored as an xml file and can be executed using the Test Framework, which is responsible for the linking of SLKs, ELKs, Adapters, etc. Figure 7 provides an overview of the current test architecture at a system level.

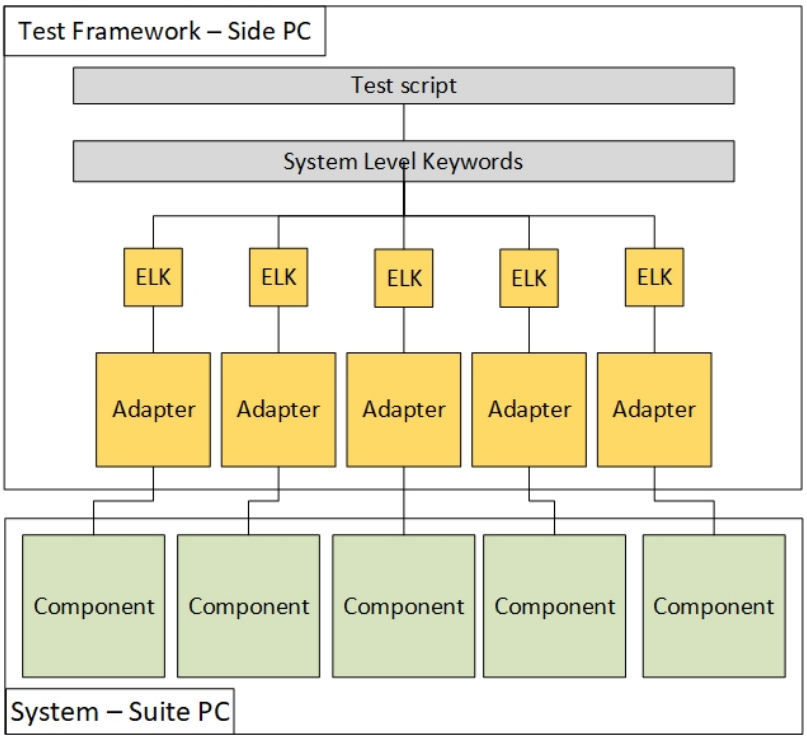


Figure 7: Current test architecture overview

The test scripts developed by the teams are stored and can be run automatically as many times as desired.

3.2 Model-based test architecture overview

In order to apply model-based testing, we designed the architecture illustrated in Figure 8. The specification given to TorXakis describes how the system behaves using the set of available keywords. With this specification, TorXakis then generates random actions, which are sent to an adaptation layer. The test cases generated by TorXakis, in this case, are composed of System-Level Keywords. For each SLK generated by TorXakis and sent to the adaptation layer via sockets in a String format, the adaptation layer is responsible for mapping the received String to the actual execution of keywords using C# code via adapter methods, which will trigger actions in the SUT via the Suite PC using the available component interfaces.

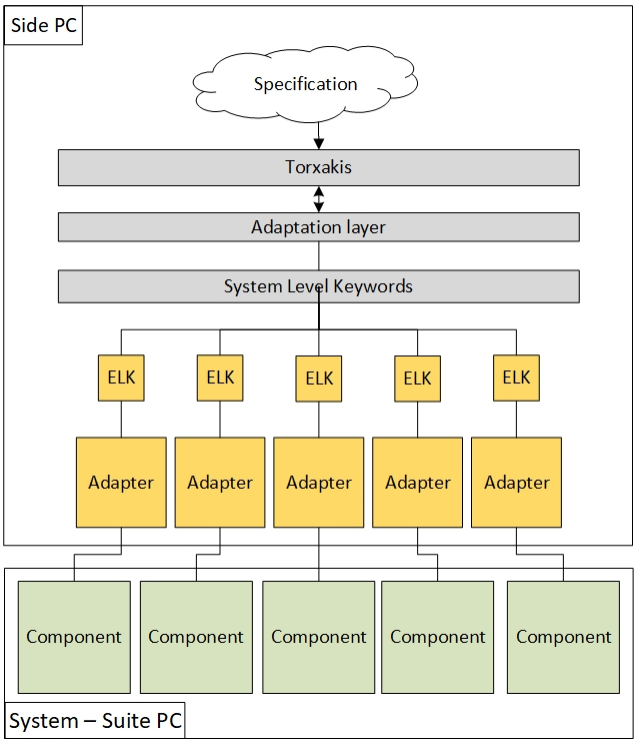


Figure 8: Model-based testing architecture

Once an action is executed, its outcome is sent back via adapters and some checks are performed in the SLKs, such as the identification of null elements. The checks performed in the SLKs are responsible for evaluate if the execution of the keyword was executed correctly or not. For example, in the keyword ActivateFluoro_Start, the inner state of the machine which indicates if the x-ray emission is active is checked before and after the execution of the keyword. The result of these evaluation checks performed by the SLKs is then sent back to TorXakis, which evaluates the received outcome and is responsible for judging whether the system behaved as intended or if the outcome was out of the specification. More details on the specification used for the model-based testing approach is given in Section 4.

The adaptation layer used in this project was implemented in C#. All the necessary link library files from the previous architecture were reused. Most of the work was in parsing the attributes for each keyword generated by TorXakis to the attributes used in the execution of the System Level Keywords.

4 Reliability testing approaches

4.1 Current approach: Automated test scripts

The current reliability testing process at Philips Image-Guided Therapy (IGT) systems makes use of automated test scripts. These scripts are made up of SLKs, which compose a predefined sequence of steps that make use of static values as input parameters. The data used in these scripts are based on data collected from physician usage in the field with regard to medical procedures. The use of real field data ensures that these systems are tested with realistic scenarios according to their intended use, which is to assist physicians in medical procedures. It is important to guarantee that the sequence of steps performed during these tests is not trivial and that a diverse sequence of steps is executed.

These test cases generated out of real field data are then used for reliability testing. The main metric used for reliability in the current way of reliability testing in a system integration level is the Mean-Time-Between-Failures (MTBF). In order to achieve the number of hours specified by the MTBF goal, the systems are tested for a long period of time, which reaches a total sum of thousands of hours. The current amount of automated scripts is not sufficient to cover the required amount of time for reliability testing, and for that reason, it is necessary to run these test cases in a loop.

The problem with using a limited amount of test cases is that as explained in subsection 2.3, software failures primarily occur when specific paths, steps, or parameters are executed in a testing run. Therefore, the execution of the same path, steps, and parameters for a long period of time does not imply an optimal reliability testing setup, unless this set of test cases would fully cover the system's behavior, which is not the case.

In order to attempt to increase the variety of test cases and provide a test set with more diversity than what is available in the current setup for reliability testing at Philips IGT systems, we propose the use of model-based testing.

4.2 Proposed approach: Model-based testing

4.2.1 General approach

Following the abstraction level applied with the current testing approach, our model makes use of keywords. Given the time constraints of this study, we selected the test case with the least amount of keywords. The chosen test case made use of 36 keywords and 842 system calls using these keywords. Out of these 36 keywords, one of them was disregarded, since its functionality was not available in the system where our tests were performed, leaving us with 35 keywords from the original set. Besides the keywords from the original test case, we added one keyword to keep track of the currently selected series of images. It was not possible to store that information in our model without that keyword since the selected series of images could keep on changing if the replay option would be activated. Different sets of keywords are available for different types of series of images, and because of that, we needed that extra keyword. With our model we could represent our system using the 35 keywords present in the selected test case and an extra keyword to keep track of the selected series of images in a given moment, resulting in a total of 36 keywords. Figure 9 illustrates an overall abstraction of the model used for our model-based approach.

The nodes illustrated in Figure 9 presents a general overview of a set of keywords. This image indicates that any keywords out of the four sets of keywords: Geometry, Acquisition, Reviewing/Viewing, and Misc can be executed. In Figure 9, each of the transitions refers to a set of keywords. GeometryKeyword refers to the set of keywords

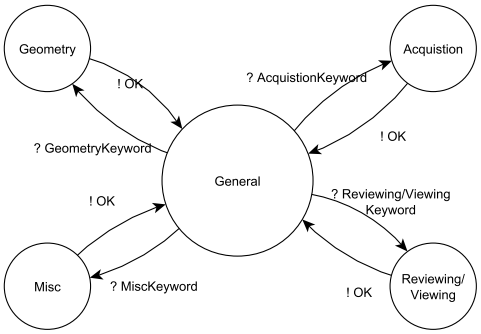


Figure 9: System abstraction model (initial state: General)

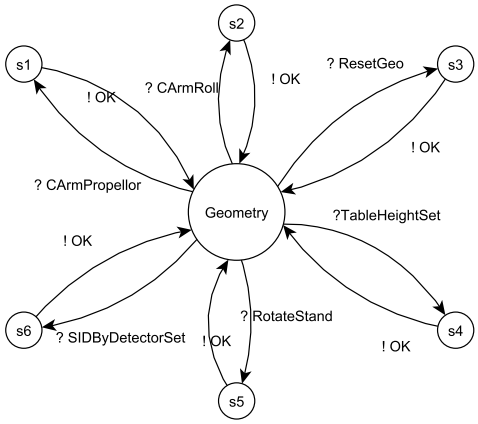


Figure 10: Geometry component abstraction model

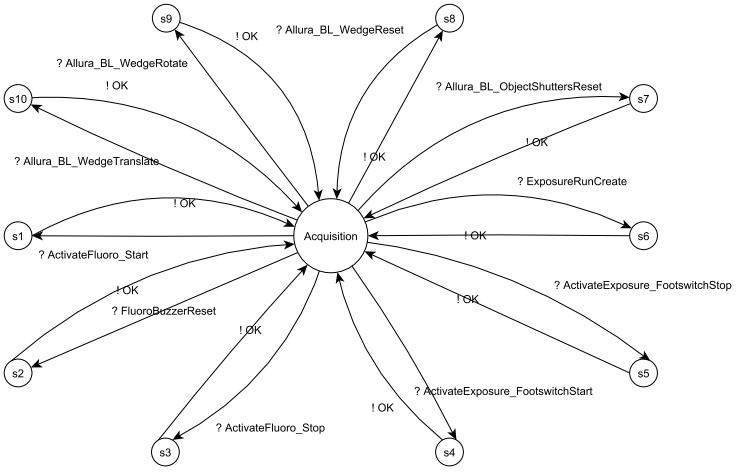


Figure 11: Acquisition component abstraction model

depicted in Figure 10, AcquisitonKeyword refers to Figure 11, Reviewing/ViewingKeyword refers to 12, and MiscKeyword refers to 13. In each of these nodes, a number

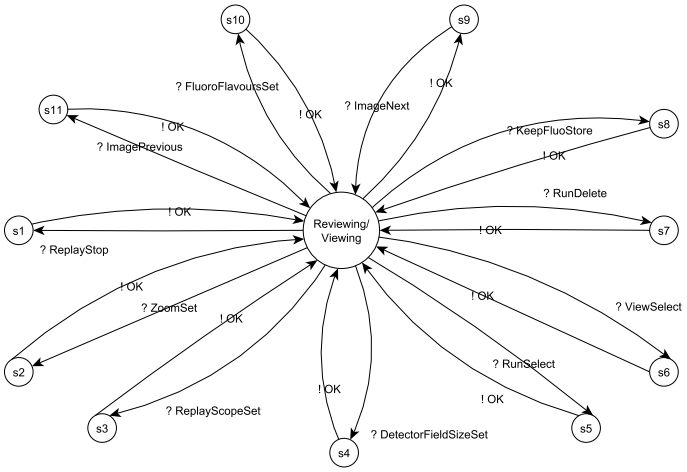


Figure 12: Reviewing and Viewing component abstraction model

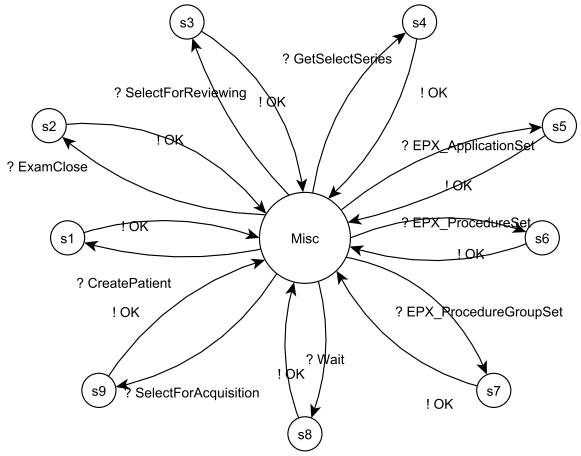


Figure 13: Misc component abstraction model

of different keywords can be found, and their general behavior can be described as the execution of an input followed by the observation of an output, this behavior is very similar to the simple model described in Section 2.3.3 (see Figure 4) and illustrated in Figures 10, 11, 12 and 13. It is important to note that some keywords can only be executed if the system is in a specific state, which is not represented in the model abstraction illustrated in these abstractions. Besides that, the observed output is not only composed by an *OK/NOK* attribute but can also be used to get information from the system. The return type used in this setup is similar to the type *SystemOutput* defined in section 2.3.3. The attribute *message* in this type is used, for example, to obtain the currently selected image key, using the keyword *GetSelectedSeries*. This key attribute can help to internally define in our model if a specific keyword can be executed or not.

With this abstraction, we make it possible for TorXakis to derive a sequence composed by any combination of the keywords used in this model. By doing so, instead of having a predefined sequence of actions, a random sequence can be derived. Besides that, most of the keywords illustrated in this abstractions make use of parameters, which are

generated by TorXakis following constraints defined by the model designer. For the sake of simplicity, these constraints are not added to the abstraction illustrations presented in Figures 10, 11, 12, and 13.

In order to guarantee that no collisions happen during the test execution and enforce that the behavior we want to test is similar to the one expressed by doctors we made use of a test purpose. In this test purpose, we made use of a predefined set of actions to be executed before the random behavior is enabled. By doing so, we can enforce that the system is in a safe position for automated testing and that a patient is selected at the beginning of a procedure, as well as the proper actions are performed when the procedure is finished. Besides that, we can also add constraints to actions, which is important since we are dealing with a system that can cause physical damage to others and itself. For this reason, actions involving movements are constrained to specific regions to avoid collisions. An abstraction of the test purpose used in this study can be seen in Figure 14, where INIT and EXIT are illustrative and not actual keywords. The number of actions to be executed for each patient N has an arbitrary initial value of 300, and this number is increased by 100 once the execution of a patient is finished.

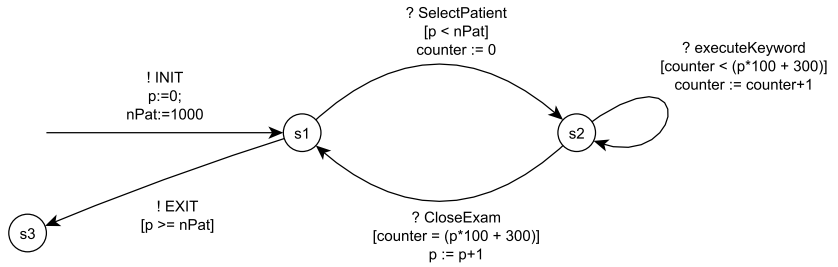


Figure 14: Abstract test purpose (initial state: s1)

With our model being 'guided' by our test purpose, we are able to derive test sequences using all the set of keywords using in the previous approach in a random sequence, including a wide range of possible parameters for each keyword execution.

4.2.2 Test case generation

In our models, all transitions labeled as an input represent a System Level Keyword and all transitions labeled as an output receives a data structure which contains information about the execution of the keyword. For each input action (keyword) generated by TorXakis, the tool receives an output from the SUT via the adaptation layer as detailed in Section 3. Therefore, given a value N of TorXakis actions, they are composed by N divided by two input actions (keywords) and N divided by two output actions.

The selection of each keyword as an input is done by TorXakis in a random manner. Each keyword has its own parameters and it is represented using a type definition (TYPEDEF), similar to the type *Keywords* defined in Section 2.3.2. Each input is followed by an output, the latter is represented by an structure using TYPEDEF, similar to the type SystemOutput in Section 2.3.2. Where *rettype* informs if the keyword execution was executed without error messages contains information on how the execution went. In case resultType is *OK*, the *message* is "No errors" by default. In case of *NOK*, all the error messages generated in the keyword execution are sent to TorXakis.

The traces generated by TorXakis are similar to the ones below:

```
TXS >>  .....1: IN: Act { { ( InputInterface, [ Keyword1(3,"oi") ] ) } }
TXS >>  .....2: OUT: Act { { ( OutputInterface, [ Return(OK,"No errors") ] ) } }
TXS >>  .....3: IN: Act { { ( InputInterface, [ Keyword1(17,"hoi") ] ) } }
TXS >>  .....4: OUT: Act { { ( OutputInterface, [ Return(OK,"No errors") ] ) } }
```

```
TXS >>   .....5: IN: Act { { ( InputInterface, [ Keyword3(-1) ] ) } }
TXS >>   .....6: OUT: Act { { ( OutputInterface, [ Return(OK,"No errors") ] ) } }
TXS >>   .....7: IN: Act { { ( InputInterface, [ Keyword2(0,"hi") ] ) } }
TXS >>   .....8: OUT: Act { { ( OutputInterface, [ Return(OK,"No errors") ] ) } }
...
```

The traces above illustrate how the testing using TorXakis was performed. A SLK is randomly chosen by TorXakis out of the set of available keywords according to the current state. After sending this keyword to our SUT as an input, TorXakis then expects an output informing if the keyword was correctly executed. This cycle of input followed by an observed behavior continues as long as desired. More details about the use of TorXakis in our experiment are given in the next section.

5 Experimental set up

In order to compare the current way of testing at Philips IGT Systems using automated test scripts in a loop and the use of model-based testing we defined a comparison study which is described in this section.

5.1 Experiment

Both approaches had to run with the same software versions in real systems in order to get accurate results. The reason behind the use of a real system instead of a simulated environment comes from the fact that on a system level, we would like to see how all the components interact together, including the embedded software from the multiple components available in a real system, which behave differently than a simulated environment, due its complexity.

Given the limitations in the availability of real systems for testing, we could not run our experiment for the necessary amount of time to achieve the MTBF goal, which would involve using multiple systems for a long period of time, resulting in a combined total amount of hours reaching thousands of hours in a row. Therefore, our experiment restrained the amount of time to 10 hours, which was possible to achieve overnight.

Even though the amount of time for this study was reduced, it was still necessary to run the selected test case in a loop in order to achieve the specified amount of time for our test run. For that reason, our setup for the current way of testing is still representative of the original testing approach.

This experiment was performed using an Azurion system, monoplane with a frontal 15-inch detector, located in the testing bay in Philips, Best. The testing covered features that are roughly depicted in Figures 10, 11, 12, and 13. They include geometry movements, image acquisition, viewing, and reviewing of images.

We recreated the test script of the original approach using a test purpose in TorXakis. This test purpose executed the same sequence of steps from the original test script in a loop, just as in the original approach. This conversion from automated test script to test purpose allowed us to:

- Provide evidence that the adaptation layer implemented for the model-based testing approach worked properly;
- Provide evidence that the model developed for this experiment did not add constraints to the behavior of the system, considering the original test case;
- Demonstrate that by using test purposes TorXakis we could recreate the current way of testing without any limitations.
- Make use of the same testing architecture for both approaches. By making use of two different architectures, we could add deviations to the experiment, which are undesirable.
- Make use of the same set of keywords in both approaches. Since there was an addition of an extra keyword used in the model-based approach to keep track of the currently selected image, we needed to add that keyword to the original test case. Even though this test purpose did not make use of the extra information, the fact that both approaches use the same set of keywords is also important not to create undesirable deviations.

As explained in Section 4, for our model-based approach we defined a test purpose in Torxakis, which limited the actions of our systems for safety but also steered the model

in the direction of more realistic use of the system (see Figure 14). In regard to safety, we enforced that only specific movements were allowed in order to decrease the risk of collisions. By doing so, we were able to generate sequences of tests consisting of an initial action of creating a patient, performing a random number of steps, and finishing the procedure.

For both approaches, we started our Torxakis tool, which was connected via sockets to the adaptation layer running in the Side PC, which in turn was connected to the Suite PC. Given the nature of model-based testing, in particular, Torxakis, once a trace is identified as out of specification, the tool stops. Since we had a time goal, we made use of a script, which could identify if Torxakis had stopped and restarted the tool to continue with the testing.

All the data generated by the test runs of both approaches were then stored, so we could extract information to perform a comparison between them. With this data, we could obtain information such as the number of bugs found, the number of unique parameter values used for each keyword, as well as the whole testing sequence.

Since both test approaches generated test runs with a similar beginning and end, we could then calculate how similar these sequences of CreatePatient $\rightarrow$ NSteps $\rightarrow$ FinishProcedure are among themselves with a test set, which is our total run for a given approach.

5.2 Evaluation metrics

As mentioned in the Introduction, we used Jaccard (J) [7], Gower-Legendre (GL) [3], and Sokal-Sneath (SS) [18] similarity indexes to measure the similarity within test sets in our experiment, based on the work presented in [4], which indicates that these are the best similarity functions in the context of similarity-based selection techniques for model-based testing.

The Jaccard similarity metric first appeared in [7], where the author performed a study to measure the distribution of the flora in the French Alps. Even though its first use was in the field of Biology, this metric has been widely used in Computing Science [24, 22, 17] and other fields to measure (dis)similarity between elements. Gower-Legendre and Sokal-Sneath are variations of Jaccard, and as expressed in Equation 1, the main difference between them is in the weight given to the differences between two sets. For Jaccard, Gower-Legendre and Sokal-Sneath we need $\alpha = 1$, $\alpha = 0.5$ and $\alpha = 2$, respectively.

$$Sim(A, B) = \frac{|A \cap B|}{|A \cap B| + \alpha(|A \cup B| - |A \cap B|)}. \quad (1)$$

In our experiment, we use Equation 1 to compare the similarity between test cases. A and B represent two sets of keywords used in two different test cases. The use of $|\ |$ indicates the number of elements in a set. Given an illustrative set of test cases (TS):

- A: RotateStand(45.5)$\rightarrow$ CArmRoll(4.5)$\rightarrow$ ActivateFluoro$\rightarrow$ Wait(5)$\rightarrow$ StopFluoro
- B: RotateStand(45.5)$\rightarrow$ CArmRoll(4.5)$\rightarrow$ ActivateFluoro$\rightarrow$ Wait(5)$\rightarrow$ StopFluoro
- C: RotateStand(15.0)$\rightarrow$ CArmRoll(8.5)$\rightarrow$ ActivateFluoro$\rightarrow$ Wait(5)$\rightarrow$ StopFluoro
- D: RotateStand(10.0)$\rightarrow$ ActivateExposure$\rightarrow$ Wait(1)$\rightarrow$ StopExposure

As an example, the calculation of the aforementioned metrics for the pair of test cases A and C would be:

$$J(A,C) = \frac{|A \cap C|}{|A \cap C| + 1(|A \cup C| - |A \cap C|)} = \frac{3}{3 + (7-3)} \approx 0.4287$$

$$GL(A,C) = \frac{|A \cap C|}{|A \cap C| + 0.5(|A \cup C| - |A \cap C|)} = \frac{3}{3 + 0.5 * (7-3)} = 0.6$$

$$SS(A,C) = \frac{|A \cap C|}{|A \cap C| + 2(|A \cup C| - |A \cap C|)} = \frac{3}{3 + 2 * (7-3)} \approx 0.2727$$

The similarity between each pair of test cases in the set (TS) defined above is given by Table 1:

Pair of test cases	**J**	**GL**	**SS**
Sim(A,B)	1.0	1.0	1.0
Sim(A,C)	0.4287	0.6	0.2727
Sim(A,D)	0.0	0.0	0.0
Sim(B,C)	0.4287	0.6	0.2727
Sim(B,D)	0.0	0.0	0.0
Sim(C,D)	0.0	0.0	0.0

Table 1: Similarity among illustrative test cases

5.3 Evaluation

As explained in Section 4, both approaches generated test cases that followed the pattern CreatePatient → NSteps → FinishProcedure. The output of each test case execution done by Torxakis was stored as a text file. Therefore, each file represented a test case.

In the calculation of the Jaccard, Gower-Legendre and Sokal-Sneath indexes, we only took into account the keywords which were used as input in our experiment, because the original test cases from the traditional way of testing did not contain expected outputs, and we expected to get always the same output composed of *(OK,"No errors")*, which would not have a significant influence on the results.

After calculating the similarity indexes for all possible pairs of test cases, we took their average value.

By using the aforementioned metrics, we expected to observe a high similarity within the test set generated by the current looping of a single test case, whereas the use of model-based testing should present a low similarity.

Furthermore, we also compared the number of bugs (B) found using each approach, as well as the number of unique parameter values (UPV) for the given set of 36 keywords used in each testing runs. This means that if a keyword is used with an n number of unique parameter values, this will add n to the total amount of unique parameters, which is the total sum of all the different parameter values for all keywords. Besides that, we also compare the total number of keywords executed (KE) in the test cases and the number of test cases (NTC) executed by each approach.

The results obtained by this experiment are described in the next section.

6 Results and Discussion

In this section, we provide information about the results obtained from our experiment. An overview of the results obtained in this experiment is presented in Table 2. Each category of the obtained results will be discussed in the next sections.

Test approach	B	KE	UPV	J	GL	SS	NTC
Model-based approach	**3**	**18334**	**4478**	**0.1453**	**0.2525**	**0.0786**	**56**
Current approach	0	14824	196	0.9320	0.9630	0.8812	16

Table 2: Overview of results.

6.1 Number of bugs (B)

In Table 3, we can observe that the model-based testing approach found three bugs, whereas the current testing manner found none. If we assume that the software release tested in this experiment has not added any new bugs to the previous software version in which the current way of testing was performed before, the outcome of no bugs being found is expected.

The reasoning behind this is that since we had already executed this predefined sequence of steps in the already existing test case, the execution path in our SUT has already been covered and a change in the outcome would be unexpected. The obtained outcome in combination with this reasoning reinforces the fact that static sequences of steps make sense when performing regression testing, but when considering software reliability, we should always aim for a diverse sequence of steps and parameter values.

Test approach	Number of bugs
Model-based approach	**3**
Current approach	0

Table 3: Number of bugs found by both approaches.

In regard to the outcome obtained by model-based testing, we can also remark that the use of diverse sequences and parameter values could unveil bugs that were undiscovered. Two of the bugs found by model-based testing covered a sequence of steps that had not been tested before and therefore was never found. The remaining bug was found by using a valid input parameter, which revealed an unexpected behavior. The diversity in the sequences and parameter values generated by our model-based approach can be observed in the next subsections.

6.2 Number of keyword executions (KE)

We can observe in Table 4 that the number of keywords executed in the test cases was similar, with a slight advantage for model-based testing. This can be explained by the diversity of the amount of time that is taken by the execution of a keyword or a set of them. For example, the amount of time taken to perform an x-ray exposure has a time attribute attached to it, which can vary. Besides that, even when there are no time-related attributes attached to a keyword, the execution time of a keyword involving movements can also vary depending on the goal and the current position of the system.

Test approach	Number of keywords
Model-based approach	**18334**
Current approach	14824

Table 4: Number of keywords generated by both approaches.

6.3 Number of unique parameter values (UPV)

In Table 5, we can observe a large difference in the number of parameter values used for the total set of keywords used in our test runs. Naturally, by running the same test case in a loop, the number of unique parameter values used by the SLKs does not increase with time, which results in a small number of used parameter values in the total run. This is not desired, since our focus is in software testing, the wider the variety of parameter values used in a test run, the better. Since the total coverage of parameter values for a given keyword is often impossible, for example, when using integers as a parameter, testers have to limit the number of values used in test cases. This choice, however, can be arbitrary and lead to problems.

An improvement on the arbitrary choice of parameter values for testing is the use of boundary testing, which can cover the critical values for a function. With model-based testing, a wide range of parameter values can be generated automatically, as demonstrated in our results in Table 5.

Test approach	Number of unique parameter values for keywords
Model-based approach	**4478**
Current approach	196

Table 5: Number of unique parameter values used by keywords in both approaches.

6.4 Similarity of keywords used within a test set

As previously discussed, our goal in software testing, in general, is to have a high diversity in the sequence of steps performed in our tests. By making use of a diversified sequence of steps, we obtain a more extensive coverage of possible states in our SUT, which is the desired goal when testing software. Therefore, we want to obtain a low similarity among test cases in order to better test.

In Table 6, we can observe that the average similarity of keywords used within the test set of the model-based testing approach is remarkably lower when in comparison to the current testing approach. This can be explained by the fact that the same sequence is executed over and over in the current way of testing. The similarity index in the current testing approach does not reach the index value 1 because the last test case was interrupted when the 10 hours of testing were reached, and for that reason, there is a small dissimilarity between test cases.

When considering the model-based testing approach, we can observe that all metrics indicate a low similarity among test cases. This provides evidence that given the randomization obtained by the model-based approached very diverse sequences are generated when testing.

Test approach	J	GL	SS
Model-based approach	**0.1453**	**0.2525**	**0.0786**
Current approach	0.9320	0.9630	0.8812

Table 6: Similarity among test cases within test set in both approaches.

6.5 Number of test cases (NTC)

As we can observe in Table 7, the model-based approach executed a higher number of test cases than the current approach. This can be explained by the fact that almost all test cases in the model-based approach ran into bugs, and for that reason, the test case was interrupted, and a new test case was started.

The results obtained in this criterion shows that in the same period of time, a higher number of diversified test cases was observed in the model-based approach. This advantage on the number of test cases is beneficial because, in the field, physicians perform a wide variety of procedures with different sequences of actions.

Test approach	Number test cases
Model-based approach	**56**
Current approach	16

Table 7: Number of test cases executed by both approaches.

7 Conclusion and future work

This thesis aimed to evaluate the benefits of a model-based reliability testing approach for Philips image-guided therapy systems. Based on the results obtained in our experiment, we can conclude that by using a model-based reliability testing approach we increased the diversity of steps, parameter values, and sequences in test cases, and by doing so we are able to find bugs that were not discovered by an approach in which the same sequence of steps and parameter values are repeated in a loop.

As discussed in this work, even though software does not have the same characteristic as hardware, there is an underlying random process in bug detection, which is purely given by the fact that the detection of a bug depends on the execution of a specific test and the occurrence of that specific test is random. Therefore, there is a need to use a random variable for reliability testing, which is usually time.

In that sense, the use of Mean-Time-Between-Failure is applicable for software testing. However, there must be a great amount of effort put into generating test cases that reflect realistic use of the system under test. This can be done through automated test scripts, but as we observed in this case study, the number of test scripts can be limited, given the need to perform long periods of time for testing to guarantee a certain level of confidence.

Can we find a better way of estimating/measuring reliability than the current setup at Philips? Or in general cases?

The use of MTBF can be used as an indicator for software reliability; however, the availability of a great number of models for reliability measurements could provide more information regarding reliability. In this paper, we mention work in the literature which provides guidelines on how to apply these models and help in the process of finding the most suitable model for specific types of application.

The combination of prediction/measurement models and the use of model-based testing can issue great improvements in the current setup for reliability testing at Philips.

Can reliability testing at Philips benefit by the use of model-based testing? If so, how?

Based on the results obtained by our experiment, we can confirm that the use of model-based testing provides great benefits to the current way of testing reliability at Philips IGT. This is supported by the evidence that shows that bugs that were not covered by the traditional way of testing were revealed when using a model-based testing approach. The variety of the test cases widely increased as well as the variability in the sequences generated by model-based testing.

In order to put the model-based approach described in this thesis in practice, Philips would need to invest resources to translate specification and requirement documents into models. Ideally, the documentation of requirements in the project definition stage should be done using models, instead of long and possibly ambiguous requirement documents, which would facilitate the use of model-based testing. Once specification models are created, the next step would be the creation of an adaptation layer to link a model-based testing tool and the SUT interfaces. In case these specification models are not created using TorXakis (or any other model-based testing tool), extra work would be necessary to convert the models into the syntax of the desired tool.

Future work should take place to investigate whether the use of model-based reliability testing provides similar benefits to the ones observed in this study in other systems or domains.

References

[1] *Torxakis*. https://github.com/TorXakis/TorXakis/wiki. Accessed: 13-AUG-2019.

[2] Bertolino, A.: *Software testing research: Achievements, challenges, dreams*. In *Future of Software Engineering (FOSE '07)*, pages 85–103, May 2007.

[3] Gower, J. and P Legendre: *Metric and euclidean properties of dissimilarity coefficients*. Journal of Classification, 3:5–48, February 1986.

[4] Hemmati, Hadi, Andrea Arcuri, and Lionel Briand: *Achieving scalable model-based testing through test case diversity*. ACM Trans. Softw. Eng. Methodol., 22(1):6:1–6:42, March 2013, ISSN 1049-331X. http://doi.acm.org.ru.idm.oclc.org/10.1145/2430536.2430540.

[5] IEC 60050-192: *International Electrotechnical Vocabulary – Part 192: Dependability*. The International Electrotechnical Commission (IEC), 2015.

[6] IEEE Std 1633-2016: *IEEE Recommended Practice on Software Reliability*. pages 1–261, 2017.

[7] Jaccard, P: *Etude de la distribution florale dans une portion des alpes et du jura*. Bulletin de la Societe Vaudoise des Sciences Naturelles, 37:547–579, January 1901.

[8] Jelinski, Z. and P. B. Moranda: *Software reliability research*. In *Statistical Computer Performance Evaluation*, pages 465–484. Academic Press, 1972.

[9] L., Frantzen, Tretmans J., and Willemse T.A.C.: *A symbolic framework for model-based testing*. In *Formal Approaches to Software Testing and Runtime Verification*, 2006, ISBN 978-3-540-49703-5.

[10] Littlewood, B. and J. L. Verrall: *A Bayesian Reliability Growth Model for Computer Software*. Journal of the Royal Statistical Society Series C, 22(3):332–346, November 1973. https://ideas.repec.org/a/bla/jorssc/v22y1973i3p332-346.html.

[11] Lyu, Michael: *Software reliability engineering: A roadmap*. pages 153–170, June 2007, ISBN 0-7695-2829-5.

[12] Masud, M., M. Iqbal, M. U. Khan, and F. Azam: *Automated User Story Driven Approach for Web-Based Functional Testing*. Journal of the Society for Industrial and Applied Mathematics, 11(1):91 – 98, 2017, ISSN 1307-6892.

[13] Mohacsi, S., 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*, pages 382–389, Aug 2015.

[14] Musa, J. D.: *Operational profiles in software-reliability engineering*. IEEE Software, 10(2):14–32, March 1993, ISSN 0740-7459.

[15] Musa, J. D. and W. W. Everett: *Software-reliability engineering: technology for the 1990s*. IEEE Software, 7(6):36–43, Nov 1990, ISSN 0740-7459.

[16] O'Connor, Patrick D. T.: *Practical Reliability Engineering*. Wiley, 4th edition, 2002.

[17] Shi, R., K. N. Ngan, and S. Li: *Jaccard index compensation for object segmentation evaluation.* In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 4457–4461, Oct 2014.

[18] Sokal, Robert R. and Peter H. A. Sneath: *Statistical Methods for Quality, Reliability and Maintainability*. 2012.

[19] Soyer, Refik: *Software reliability.* WIREs Comput. Stat., 3(3):269–281, May 2011, ISSN 1939-5108. `https://doi.org/10.1002/wics.159`.

[20] Timmer, M., Brinksma H., and Stoelinga M. I. A.: *Model-Based Testing*.

[21] Tretmans, J.: *Formal methods and testing.* chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 3-540-78916-2, 978-3-540-78916-1. `http://dl.acm.org/citation.cfm?id=1806209.1806210`.

[22] Vernica, Rares and Chen Li: *Efficient top-k algorithms for fuzzy search in string collections.* In *Proceedings of the First International Workshop on Keyword Search on Structured Data*, KEYS '09, pages 9–14, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-570-3. `http://doi.acm.org.ru.idm.oclc.org/10.1145/1557670.1557677`.

[23] Whittaker, James A. and J. H. Poore: *Markov analysis of software specifications.* ACM Trans. Softw. Eng. Methodol., 2(1):93–106, January 1993, ISSN 1049-331X. `http://doi.acm.org.ru.idm.oclc.org/10.1145/151299.151326`.

[24] Yadav, Sandeep, Ashwath Kumar Krishna Reddy, A.L. Narasimha Reddy, and Supranamaya Ranjan: *Detecting algorithmically generated malicious domain names.* In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 48–61, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0483-2. `http://doi.acm.org.ru.idm.oclc.org/10.1145/1879141.1879148`.

[25] Özekici, S and Refik Soyer: *Reliability of software with an operational profile.* European Journal of Operational Research, 149:459–474, September 2003.