Radboud University Nijmegen

Master thesis, 2015.

Model-based robustness analysis of indoor lighting systems

August 2015

Author:

Danny Hendrix

Radboud University:

Supervisor:
Dr. ir. G.J. Tretmans

Assessor:
Prof. dr. J.J.M. Hooman

TNO-ESI:

Dr. J.H. Verriet

Dr. ir. R.M.P. Doornbos

Abstract

In this thesis, the robustness of indoor lighting systems has been analysed. To define robustness in terms of testing, a categorization in robustness of system engineering is proposed. The categorization divides robustness by means of characteristics in input and consequences of a changing environment or internal structure.
Testing robustness of software is difficult and requires a different approach than testing normal behaviour. Preferably, testing is fully automated including the generation of test cases. One automated approach is model-based testing. In model-based testing, a formal model describes the behaviour of the system. The model is used to mathematically derive tests to validate whether an implementation is a valid implementation of the model. In addition the formal model can be used to reason about the behaviour of the system. A particular technique is model checking. In model checking, properties define the allowed behaviour of the system. With model checking, proof can be given whether a property holds in the model or not. The combination of model checking and model-based testing allows a model driven approach to system development where faults in the design are detected in an early stage. For each of the proposed categories in robustness, a method is provided to test robustness with a model-based approach; that is, model checking combined with model-based testing. Experiments have been performed where model-based testing is applied to test indoor lighting systems. Both the normal behaviour as well as behaviour after message loss is considered. The results show that model checking is best to find inconsistent behaviour, whereas model-based testing provides confidence in the conformance relation between the model and the implementation. Due to the limit in state coverage, model-based testing did not provide new insight considering message loss. It did provide insight in the correctness of the model.

Table of contents

1 Introduction

Testing is an important part of system development. Testing in general is a difficult, expensive, time-consuming and labour-intensive process [31]. Preferably, testing is done automatically, allowing to repeat the tests each time the system is modified. One of the automated testing techniques is model-based testing. In model-based testing, a formal model of the system is constructed from the specification. This allows to perform reasoning, such as model checking, to validate the correctness of the model.
With the use of model-based testing, the model can be used to test whether an implementation under test (IUT) is conform to the model, and thus conform to the specification if the model is a valid representation of the specification. The model specifies when and what inputs and outputs the IUT should accept and produce. Model-based testing has shown to be an effective method for testing embedded systems [34].
One of the more difficult aspects of systems to test is robustness. In robustness testing, the system is tested against an unreliable environment or unexpected inputs. Robustness testing is difficult because it is hard to specify robustness properties. Many different definitions of robustness have been given in the literature, ranging from recovery after unexpected inputs to resilience of the system in an unreliable environment.
In this thesis, a categorization of system robustness is proposed. In addition, a model-based approach, that is, model checking and model-based testing, is proposed for each of these different categories. The theory is then applied to model-based test robustness of indoor lighting systems.

1.1 Prisma project

Modern indoor lighting systems consist of much more than just a set of lamps. Motion and daylight sensors support the luminaire in deciding where and when to produce the right amount of light. Currently TNO-ESI is collaborating in the Prisma project with Philips Lighting. The aim of this project is to analyse and improve the robustness of indoor lighting systems. In this thesis I will contribute by applying model-based testing techniques to further analyse the robustness of such smart lighting systems.
One of the interesting aspects in the analysed lighting systems is message loss. Luminaires communicate with each other when, for instance, one luminaire detects motion and all the luminaires should go to the corresponding light level. The luminaires should be robust against possible loss of messages. Message loss can result in the system being in an inconsistent state. An inconsistent state can for instance be a situation where all luminaires in a room are on except for one, because this luminaire has missed a message. Some of the inconsistent situations are not much of a problem. For instance, it is not

a problem if one luminaire is more dimmed than the other luminaires for a couple of minutes. A more critical problem is, for instance when a luminaire never switches off! Philips Lighting would like to have an indication of these inconsistent situations and whether the system recovers within a specified time span.

1.1.1 Prisma models

TNO-ESI has created models of the behaviour of the system. In the remainder of this thesis, these models are referred to as Prisma models. With the use of model checking a number of possible inconsistencies, caused by message loss, have been detected. The failure scenarios that were found with model checking have been reproduced on the actual system by filtering out the appropriate messages [12]. The correctness of the model that has been used in model checking, is formed from the specification and has been validated by means of observing the behaviour on the actual system. It is possible that automated model-based testing techniques can find differences between the Prisma model and the actual system. Part of this thesis is the online verification of the models with a small test environment. Applying model-based testing with the Prisma models, that are used for model checking, will give insight in the added value and practical limitations of model-based testing combined with model checking.

1.2 Research aim

The aim of this research is to define the different types of robustness in system engineering and explore how model-based testing can be used to test robustness in an automated way. Some of the proposed methods will then be applied to test robustness of indoor lighting systems.

The research question that will be studied in the master Thesis is stated as followed:

What are the different types of robustness in system engineering and can they be tested using a model-based approach, furthermore, is model-based testing a useful method to test robustness of indoor lighting systems?

Questions that derive from the main research question are:

1. What is considered robustness in system engineering?
2. How can robustness be categorized in terms of testing?
3. How can model checking be used to analyse robustness?
4. How can model-based testing be used to test robustness?
5. What are the strengths and weaknesses of model-based testing compared to model checking for validating robustness of a system?

6. Is the Prisma model a good representation of the real implementation?

7. Which model-based testing tools can be used to test indoor lighting systems?

8. To what extend has the Prisma model, used with model checking, be changed to make it usable for model-based testing?

9. How can robustness against message loss be tested with model-based testing?

10. How can robustness against power loss be tested with model-based testing?

1.3 Related research

In [6] a framework is proposed for assessing robustness. The authors propose a method to calculate the total risk probability in system engineering. Their method addresses robustness by means of probability of exposures, that lead to damage which in turn lead to consequences in the behaviour of the system. The research considers direct risk, which is associated with risk that leads to potential damages in the system, and indirect risk which corresponds to increased risk of a damaged system [6]. Test cases are often selected on risk analysis. When creating categories in robustness with respect to testing, risk of damaged systems should preferably be included.

There has been research in the field of model-based testing [11]. Model-based testing has been applied frequently to engineering systems to test the normal behaviour of the system. Larsen et al. have applied model-based testing with the tool Uppaal Tron in an industrial environment [19]. The used model is a timed automata network in Uppaal. The model acts as a specification of the system that is tested against the real system with Tron. Their research did not find systems errors. The authors argue that the system that is being analysed is stable and has been used in the field for many years. The authors believe that their approach will find system errors in newer, less-tested software. One of their conclusions is that most of the inconsistencies between the model and the implementation under test (IUT) were caused by incorrect models, caused by incomplete specification documents.

Methods of automated checking, using a formal specification, include model checking. Research has shown that model checking is a successful method for finding software bugs in embedded systems. Within the Prisma project, model checking proved to be a successful method for detecting inconsistent states in lighting systems [12].

In model checking, the model represents the IUT. To validate the correctness of the model, a set of properties are derived from the specification. A model checking tool can then verify whether the properties hold in the model [10]. In model-based testing, the model is the specification and is tested against the real IUT.

In many research, automated testing and model checking is combined by using the failure scenarios found with model checking and execute them on the IUT [3, 4, 12]. This makes the assumption that the model used in the model checking phase is a correct representation of the real system.

In [13] a framework for model-based testing of robustness is introduced. The proposed framework is very theoretical and has only been applied to small toy systems. In this thesis, the system that is being analysed is much larger. Furthermore, lighting systems are highly depending on time as light should automatically go off after a period of not detecting motion. The proposed framework in [13] does not consider timed specification of the IUT. Another limitation of the proposed framework is that it mainly focusses on fault-injection; robustness can be seen much broader than just fault injection. What is currently missing in the literature is a good and broad categorization of robustness in engineering systems.

1.4 Structure of the Thesis

Chapter 2 describes different definitions of robustness and proposes a categorization of robustness to answer question 1 and 2. Chapter 3 describes how model-based testing can be used to test the different types of robustness and answers question 3, 4 and 5. Chapter 4 describes the lighting system that has been used to apply model-based robustness testing. Chapter 5 describes the model-based testing experiments that were performed and answers question 6, 7, 8, 9, and 10. Chapter 6 evaluates the experiments and discusses advantages of model-based testing against model checking and discusses whether model learning can be of use within the Prisma project. Finally Chapter 7 concludes the thesis and provides suggestions for future work.

2 System robustness

This chapter describes the different types of system robustness and proposes a categorization. First a number of different definitions of robustness are given that are used as a basis for the, to be formed, categories. Categories are explained in terms of examples. In the next chapter, where a model based approach is proposed, categories are formalized. To illustrate robustness problems, vending machines rather than lighting systems are used as examples. Vending machines are more diverse and the behaviour is assumed to be more familiar to a larger audience.

2.1 What is robustness?

The term robustness is not limited to system engineering but definitions in other disciplines are applicable to system engineering. Many definitions overlap in their meaning. The Santa Fe Institute has formed a list of different definitions of Robustness, formed from different disciplines [15, 24]. The following list gives a set of definitions for robustness, that are relevant for system engineering. Some of these definitions are not targeted at system engineering but they do contain aspects that are relevant. The list is constructed from the list of definitions of the Santa Fe Institute, extended with definitions found in the literature.

R1 Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [1].

R2 Robustness is the persistence of specified system features in the face of a specified assembly of insults [2].

R3 Robustness is the ability of a system to maintain function even with changes in internal structure or external environment [9].

R4 Robustness is the ability of a system with a fixed structure to perform multiple functional tasks as needed in a changing environment.[17].

R5 Robustness is the degree to which a system is insensitive to effects that are not considered in the design [29].

R6 Robustness is the ability of software to react appropriately to abnormal circumstances (i.e., circumstances “outside of specifications” including new platforms, network overloads, memory bank failures, etc.). Software may be correct without being robust [21].

R7 Robustness is a design principle of natural, engineering, or social systems that have been designed or selected for stability [15].

R8 Robustness is a characteristic of systems with the ability to heal, self-repair, self-regulate, self-assemble, and/or self-replicate [15].

R9 Robustness can be defined as the capability of a control system to remain working correctly and relatively stable, even in presence of disturbances. Additionally, an important issue is the system fault tolerance, i.e. the capability to detect and tolerate internal failures, in order to continue performing their operations without the need for an immediate intervention [20].

R10 Robustness is a measure of feature persistence for systems, or for features of systems, that are difficult to quantify, or to parametrize [17].

R11 Robustness is a measure of feature persistence in systems where the perturbations to be considered are not fluctuations in external inputs or internal system parameters, but instead represent changes in system composition, system topology, or in the fundamental assumptions regarding the environment in which the system operates [17].

R12 Robustness can be defined as the ability of software to keep an "acceptable" behaviour, expressed in terms of robustness requirements, in spite of exceptional or unforeseen execution conditions (such as the unavailability of system resources, communication failures, invalid or stressful inputs, etc.) [13, 30].

R13 Systems that are robust often are required to maintain function while exploring new functionality [17].

What all definitions agree upon is that robustness of a system refers to the functionality of a system and is related to invalid *input* (**R1**, **R2** and **R9**) and changes in the *environment* (**R1**, **R3**, **R4** and **R11**) by which the system is surrounded. The terms input and environment can be interpreted differently. I will define what is considered to be a system, environment, and input in this thesis in Section 2.2.
One of the aspects that not all the definition agree upon is whether robustness is part of the design or not. Definition **R2**, **R7** and **R12** consider some sort of specification or requirements, related to robustness, in the design. In definition **R5**, **R6** and **R11**, robustness is the behaviour outside the design. If we consider Definition **R1**, the system should function correctly when operating under stressful environmental conditions. Correct behaviour is not necessarily normal behaviour. For example if in a vending machine the output arm (the mechanical part that picks the correct product) breaks, the machine might display an error instead of allowing customers to purchase products. In this case correct behaviour in unexpected conditions, should be specified in the design. Nevertheless, robustness is not a yes or no question. A system can only be robust to a certain level. The size of all the available inputs and the size of the environment of a system is very large, and often even infinite. We cannot make a system robust to

everything but we can say that it is robust against certain circumstances. If robustness is considered in the design, we can give a verdict whether the system is robust according to the specification. This does not mean that the system is robust in general, for that we also have to consider behaviour outside the design.
An interesting aspect of robustness is extensions to the system, as mentioned in Definition **R13**. The system should maintain functionally when new features are added to the system. This is a property of robustness that is not considered in most of the other definitions. For instance, the definition given by the IEEE (Definition **R1**) considers the system to be stable and not change functionality. With larger systems, elements of the system may be replaced with newer versions or extended with new functionality. The remaining elements should still maintain there own functionality. This is especially the case with self-learning systems where functionality continuously changes. Newly learned behaviour should not interfere with existing behaviour.

2.2 System, environment and input

The previous section provided general, not particularly related to system engineering, definitions of robustness. This section defines what is considered a system, the environment of the system, and what is considered an input of a system in this thesis.

The meaning of a system can be very different in different disciplines. The focus in this thesis is towards systems that contain software. I use the definition of a system given in NASA Systems Engineering handbook [18]:

Definition 1. *A system is a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents; that is, all things required to produce system-level results. The results include system-level qualities, properties, characteristics, functions, behaviour, and performance.*

In addition I consider only systems that contain some form of software. This can range from firmware to higher level software, such as Java applications.
In the remainder of this chapter, the word element is used and refers to an element as given in Definition 1.

Simple said, the environment of the system is everything that is not part of the system. In order to define what is considered to be the environment, the system has to be precisely defined.
I define the environment of the system as followed, where elements refer to the elements defined in Definition 1:

Definition 2. *The environment of a system is the circumstances, conditions, or elements by which the system is surrounded.*

Input of a system is everything that can be provided to the system. This means that input include, valid input that the system expects but also invalid input that the system does not expect. In the case of a vending machine, input consists, among others, of coins, button presses and power. More precise, coins can be of any currency, including currencies that the vending machine does not accept. Furthermore, input can also be observed by the system. An example is a heat sensor that measures the current temperature.

I define input of a system as followed:

Definition 3. *Input of a system is everything that is fed to, or observed by, the system.*

It is difficult to separate between what is part of the system and what is part of the environment. In the given definitions, it is not clear where the system ends and the environment begins. For example, in the case of a vending machine, the buttons can be seen as part of the system because the buttons are required to produce result and, according to Definition 1, the system consists of all elements that together produce a result that is not obtainable by the elements alone, and thus the panel is part of the system. On the other hand, if we consider the software of the vending machine, the buttons can also be seen as part of the environment. The software observers the press of a button by means of an event being triggered. The buttons are in that case not involved in the process to obtain the result and therefore part of the environment.
To define what is the environment and what is part of the system, I propose to look at the result that the system should achieve. Consider a vending machine with one button that, when pressed, outputs chocolate. If we are interested in the physical output of chocolate, the system consist of the button and all the internal elements in the machine, such as the mechanical arm that outputs the chocolate, that contribute to the process. All other elements in the machine, that do not take part in the process of producing chocolate, are not part of the system. This means that an additional button on the candy machine that outputs gum, is not considered part of the system, when only considering chocolate to be the result of the system. This additional button however is part of the system when we consider the output of all candy to be the result of the system.
Considering just the result however is not enough. According to that statement, the power source, like a power plant, also becomes part of the system because the hardware parts in the vending machine require power.
This can be avoided by only consider an element to be part of the system when the design of the behaviour of the element is in control. In the case of the vending machine, where the design of elements of the physical machine are in control, the power plant is not part of the system because the design of the power plant is not in control by the supplier of the vending machine. In addition I consider entry points of the system. To define an entry point, I consider all the input that is required to produce the result. An entry point is the latest point that receives an input before it is processed. Entry points allow to close the system when the system is part of a larger system. An example is a vending machine where both the hardware and software are from the same supplier. A

system can be defined in terms of both hardware and software but also as just software. The entry points make it possible to define both these systems. A practical approach to define the system is shown, by means of a flow diagram, in Figure 2.1.
The starting point considers no elements in system **S** and no elements as entry points in **SE**. For each output **u** in **Lu**, where **Lu** is the set of outputs that is considered in the system, **provide(u)** is the set of elements that provide output **u**. Each of these elements are part of the system, with the condition that the design of the element is in control **designInControl(s)**, and therefore added to **S**. Each of these elements requires zero or more inputs in order to produce the output. These inputs, of an element **s**, are the set **required(s)**. An element **s** is an entry point when one of the inputs is in the set of required inputs **LRI** of system **S**. If the input is not a required input, than all the elements that provide this input to **s** are part of the system (unless the design of the element is not in control) and should be analysed to further extend the system. **provide(k,a)** is the set of elements that provide input **k** to element **a**.

To illustrate, recall the vending machine with one button that outputs chocolate. Power can be defined as one of the required inputs to produce the result of the output of chocolate. This means that **LRI** ={Power,Button press} and **Lu**={chocolate}. The elements that are required to output chocolate is the mechanical part, to simplify, let us consider this to be a motor, that outputs products. **provide(u)** = {motor}. The motor is now part of the system. The input that the motor requires is power and a signal to output chocolate. Thus **required(motor)** = {power,signal}. Power is in **LRI** and therefore the motor is an entry point of the system. The second input that the motor needs is a signal. This signal is given by the hardware/computer of the vending machine. This computer in turn, requires the software and signals from the button. This makes the button and the software part of the system. When all elements are reached, the definition of the system is complete and closed.

To show that this method is also applicable to software systems, consider a Java program that consists of a user-interface with one button and one text field that displays a numeric value. Pressing the button increases the value by one. Let us consider the result of the system to be the change of numeric value and the required input to be the button being triggered. When the user activates the button, the first element that is involved is the Java application. The Java application in turn makes use of functionality in the Java Virtual Machine (JVM). The design of the JVM is not in control and therefore the JVM is not part of the system. Any element after the JVM, such as the operating system is not part of the system. The system only consists of the Java application.

There is no guarantee or proof that the described method is applicable to all systems. More research is needed to explicitly define a system and its environment.

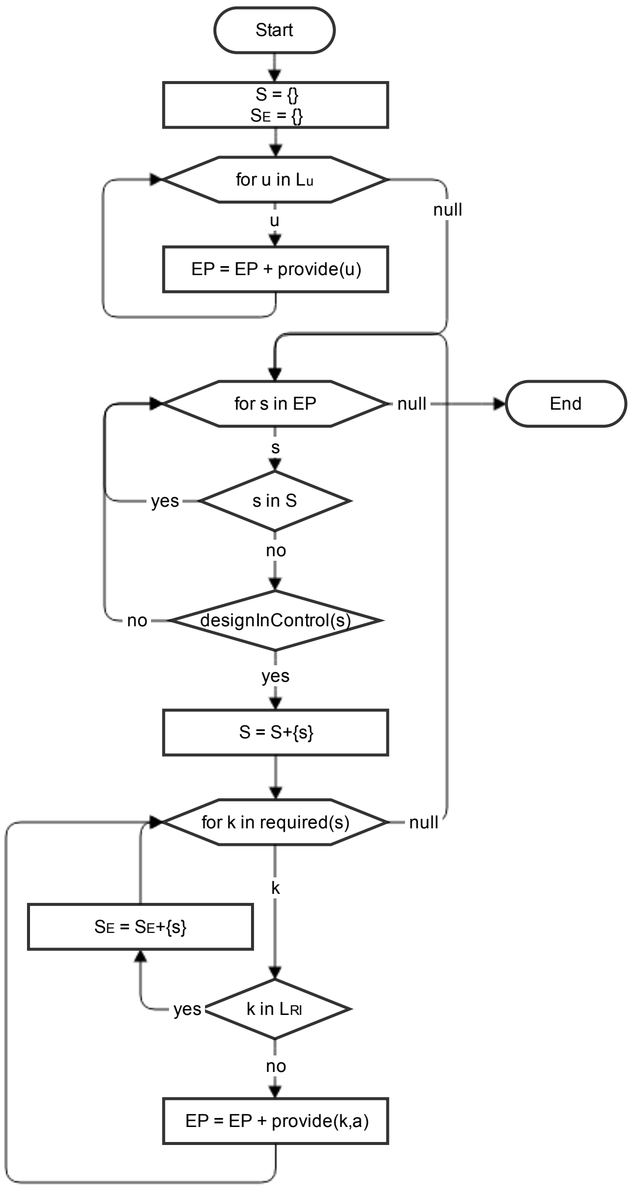


Figure 2.1: Flow diagram to define a system

2.2.1 Quality of systems

The goal of software testing is to ensure quality of software. Testing robustness can therefore be seen as validating the quality of the system in terms quality characteristics, related to robustness. The ISO 25010 standard describes a model which categorizes the product quality into characteristics and sub-characteristics [16]. Robustness is not explicitly mentioned, robustness can be seen as a second degree or underlying quality attribute [27]. The model consists of two main categories, software product quality, and software quality in use. The latter describes how usable the system is, which includes topics like, usefulness, flexibility, and pleasure.
Robustness is only considered in the first category, product quality. This category consists of eight characteristics, namely, Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, and Portability. These categories are in turn divided in sub-characteristics. When considering the definitions of robustness, I list the following quality characteristics and sub-characteristics that describe quality of a system in terms of robustness:

Compatibility

- **Co-existence** Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.

Reliability

- **Availability** Degree to which a system, product or component is operational and accessible when required for use.
- **Fault tolerance** Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
- **Recoverability** Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

Maintainability

- **Modularity** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Modifiability** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.

Portability

- **Adaptability** Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.

What is interesting to see is that robustness is part of four out of eight main characteristics of software quality. This indicates that robustness is an important part of the quality of systems which indicates that, to test robustness, different approaches are needed. For instance testing co-existence requires testing the system when operating alongside other systems whereas recoverability requires testing behaviour after unexpected input or environmental conditions. It is very unlikely that these quality aspects can be tested with one general approach. Nevertheless, some characteritics, such as co-existence and adaptability, might be testable with the same approach. Take for instance calculation time. In co-existence, hardware has to be shared with other systems which can result in slower calculation. In adaptability, hardware is improved and calculations can be executed faster. Both affect the calculation timing in the system and thus a general approach might be applicable.
To categorize robustness, we should not use quality characteristics as basis but rather use them to validate whether the categorization covers all quality characteristics, related to robustness.

2.3 Characteristics in system robustness

This section explores how different aspects in system robustness are characterized. These characteristics are used to categorize robustness in the next section.

From the different definitions of robustness, we can consider three main aspects, namely, invalid input, unreliable environment, and changes in the internal structure. In short, invalid input of a system is all unexpected input that is provided to the system, the unreliable environment is everything outside the system that affects its behaviour, and the changes in the internal structure consist of changes and extensions in the internal structure of the system. In the remainder of this section, different characteristics are identified within these three aspects.

2.3.1 Invalid input

Input of a system can have many forms. Input can range from an event, such as a button press, to continuous input, such as power. Furthermore input can be a physical object, like coins in a vending machine, or data such as the temperature of a heat sensor.
In this section I list characteristics of input. In robustness, only invalid input is of interests. Nevertheless, most of the given characteristics apply to both valid and invalid input. The distinction between valid and invalid input is considered in the content of the data, which is one of the listed characteristics.
When considering testing, not all characteristics need a different approach. For each characteristic arguments are provided why it does or does not need to be considered as a different category in robustness testing.

Data and non-data input

Input can be distinguished between data and non-data. Data input can for instance be a string, entered in a form, but can also be something like the temperature, in the form of an integer, that a heat-sensor measures. Non-data input can for instance be power or physical materials, like coins in the case of vending machines. Characteristics of non-data input is that quality and quantity become a factor. In the case of power, the system often requires a specific voltage to operate under. If the power supply cannot produce a constant power supply, the system may stop functioning. In non-data, quality of input is also an important factor. A damaged coin is different from a non-damaged coin. The vending machine should remain functional when a damaged coin is inserted. In data, a damaged coin is no different from a non-damaged coin, the value of a coin of 1 Euro, is 1 Euro, no matter if the physical coin is damaged or not.
The proposed categorization focusses on robustness of system software. Non-data input is something that the system can receive but is not directly fed to the software. To illustrate, consider inserting a coin in a vending machine. A mechanical sensor detects what the value (1 Euro, 50 cents etc.) of the coin is. The software of the vending machine receives the value that the sensor reads, which is data. Nevertheless, invalid non-data input can have effects on the software of the system. For instance, consider we insert a fake coin that is accepted by the vending machine as 1 Euro. This means that the coin is now inside the machine. When the machine has to give change, it is possible that the machine has to output a coin of 1 Euro. Let us assume that the machine performs a second check to validate the coins when they are given as output. If the fake coin does not pass the second check, the software has to respond properly to the unexpected behaviour in the mechanical part of the machine. Indirectly, this is a result of invalid non-data input of the system. Because we consider robustness with respect to software, the given example can also be considered to be caused by an unreliable environment of the system. In one of upcoming sections we explain robustness in terms of environment. We will see that these kind of failures are covered by environment characteristics. For other non-data input, like the temperature of a heat-sensor, non-data is converted to data before it reaches the software. We can therefore conclude that this characteristic does not need to be considered explicitly in the categorization for testing.

Observed and direct input

A distinction can be made between input that is fed to the system and input that is observed by the system. To avoid confusion I shall call these *direct* and *observed* input respectively.
With direct input, all input is considered that would not have been observed if an other element would not explicitly inform the system. An example is a vending machine with one button that gives chocolate. The system responses to input caused by an external element, the user, to produce output after the user has pressed the button. The user has to inform the system.
Observed input, is considered to be all the input that the system can observe without

the need of external elements informing the system. An example is a heat-sensor that measures the temperature, by means of polling, in a vending machine to avoid melting of chocolate. In this case, the system can detect the temperature periodically and does not need an external element to start interaction and produce an input.
A boundary case is input that is produced by observing an element outside the system. For example, a motion sensor will detect motion when a person moves in front of the sensor. In this case the element (user) takes action and the system observes the action. We consider this to be an observed input because the sensor is observing motion periodically and does not get activated when the person is in front of the sensor.

In testing, direct input is much easier to test because the tester can simply produce input and feed it to the system under test. In the case of observable input, the tester has to change the environmental conditions in which the (test-) system operates. For example, when testing a system that relies on a heat sensor, the tester should change the temperature in the area in which the sensor is located. Observable input can be converted to direct input. To test a system with a heat sensor, a simulated heat sensor, that is controlled by the tester, can be used to simulate different temperature measurements. However, the cost of this is that the system that is being tested is not operating under the same conditions as to when it is deployed, because the deployed system makes use of a real heat sensor. To keep the categorization practically usable, and the fact that observable input can be converted to direct input, we do not need to consider observable input explicitly in the categorization. We should focus on direct input.

Continuous input and event-based input

There are typically two forms in which input arrives at the system. There is continuous input, like power, and event-based input like a press on a button.
In testing, inputs and outputs are usually considered events. Continuous input can be converted to event-based input by means of triggering the input when the value changes. In the example of power, input events can be switching power on or off.
When considering direct and observed input, observed input is considered to be continuous input where the system should be able to observe input at all times.
Because we can convert continuous input to event-based input, any testing approach with event-based input can also be used with continuous input.

Content of input

In the content of the input we can distinguish between valid input and invalid input. In robustness testing, only invalid input is of interest. Invalid input is typically input that is not within the domain of the input of the system. However invalid input can also be input that is within the domain but the time or order of the input is not what the system expects. In network protocols, the latter is often referred to as *inopportune*, which shall be used in the remainder of the Thesis.
We define three categories in the content of the input:

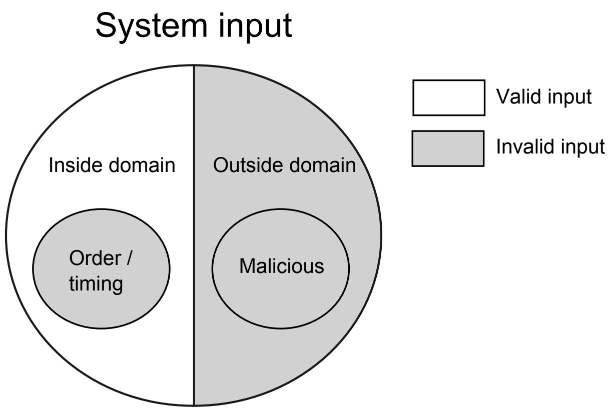


Figure 2.2: Contextual types of system input

- Inopportune or lack of input
- Input outside the domain of the system input
- Malicious input

Figure 2.2 shows how these different categories relate to one another.

The first category consists of testing of input within the domain but not at the expected time or order. The system is considered to be robust if it remains functional in unexpected circumstances which includes inopportune input. Furthermore, it is possible that the system expects input but does not receive it. This is in particular the case with continuous input, like power. The system expects to receive power at all times, but sometimes power can not be provided due to, for instance, environmental circumstances like a failure in the power network.
Testing inopportune input is a different type of testing than testing input outside the domain. Testing input outside the domain requires some form of data generation whereas testing order and timing is a matter of introducing delays or feeding valid input to the system when the system does not expect it.

Within the input outside the domain, malicious input is a category of its own. The goals of malicious input is to: disrupt or halt service, causing denials of service, access confidential information, or improperly modify the system [5].
The reason for making a distinction is that, input outside the domain is usually infinite and can thus be tested only to a limited amount which is often chosen at random. Malicious input is not random input but often contains some information from the domain and can therefore be tested is a more structured way. A good example is testing web-forms where malicious input is a string containing a SQL command, to test against SQL-injection, and input outside the domain is a very long random string, to test whether

the system rejects the long string. Both test the robustness of the web-form but different aspects are tested.
A special kind of input is corrupt input. Corrupt input is typically the result of unexpected events or failures in the environment. Corrupt input is essentially input outside the domain. We therefore do not give corrupt input a different label.

In testing we do need different categories to test different content of input. As mentioned before, testing input outside the domain as well as malicious input, require different approaches than testing inopportune input.

2.3.2 Unreliable environment

If we recall the definitions of robustness, the system should remain operational when operating under stressful, unreliable or changing environmental conditions. For clarity we use the term unreliable.
The size of the environment is very large. Due to all kinds of dependencies, a small change in the environment may have an effect on the behaviour of the system. In addition, changes in the environment may have an unexpected effect on the system's behaviour. For example, in the case of software, the software might be affected, indirectly, by the temperature of the room, in which the computer which runs the software, is located. A possible effect is slower calculations due to overheating of the computer chip.
This thesis is in the field of computer science and therefore we are only interested in the robustness of the software of the system. Nevertheless, the software should be robust against failures in other elements, such as hardware. In the environment, as well as internal structure, we could make a distinction between software and non-software. If we consider testing, unreliable software is often easier to simulate than other elements. For example, simulation of power failures, requires physical hardware to interrupt the power supply to the system. In software often a stub can be used to produce unexpected behaviour. Nevertheless, the effects of failing software or hardware can be the same. Consider a vending machine with a mechanical arm to output products and a piece of software that decides when this arm receives power. It does not matter whether the software or the power cable fails to deliver power, the effect is the same. Furthermore different failures in the environment lead to the same failures in the system. For example, power from a power plant, goes through a whole infrastructure before it reaches the system. An interruption can take place at any point in this infrastructure. In robustness testing, the cause of the power failure is not relevant and we can simulate power loss at an entry point of the power to the system.
We should not look at the cause of the unreliable environment but rather at the effects. This allows a more general approach and allows abstractions in the test setup.
I consider the following effects of an unreliable environment that lead to unexpected output or behaviour of the system.

Loss of elements

Loss of elements occurs for example if an element of the system breaks down. There are many reasons for an element breaking down due to unexpected changes in the environment. For example, a computer chip can overheat when exposed to extremely high temperatures. The makes the assumption that the computer chip is part of the system and the temperature is part of the environment.

Corrupt, lost, or unexpected input

Corrupt or loss of input takes place when the environment interferes with the system's input. In the case of a vending machine, this can be for instance when someone unplugs the power cable. The machine will no longer receive a required input (power).
Corrupt and unexpected input is essentially the same as input outside the domain which is covered in the input category in Section 2.3.1. Loss of input is also handled in the input category.

Interference in behaviour

It is possible that an element in the system gets interfered with. This can be the case when, for instance, a fault is inserted in the software or hardware to prevent the element from normal operation. An example is replace the subtract in $a = a - 1$ by addition $a = a + 1$. Interference is not necessarily caused by an attacker. Interference might as well be caused by hardware or software failures. Software might for instance, overwrite the source code.

Corrupt, lost, or unexpected (internal) data

Systems often contain data. It is possible that data is lost or altered due to events in the environment. A good example is a database that a system **A** shares with another system **B**, which is therefore part of the environment of **A**. If system **B** makes a change in the database, system **A** might be affected by it. Data loss may also occur with data that is only used by the system itself. Consider a database that is hosted on a hard drive. Placing a magnet against the hard drive can result in unreadable or corrupt data. In both examples the data is corrupt but the effect on the system is the same. As mentioned before, in robustness testing the cause of the failure is not important but the response of the system is.

Corrupt, lost, or delayed communication

A system can consist of more than one element. Different elements communicate with one another by some sort of medium. In addition, this medium might be shared with elements outside the system. It is possible that this medium cannot deliver all communication at all times. This can for instance be the case when the load is too high or a loose contact in one of the cables. Possible results are that the communication is delayed, corrupt or does not reach its destination at all.

Loss or gain of time

Computer systems require hardware to compute calculations. It is possible that the hardware cannot execute all calculates when the system needs it. In the case of computer software, other software might use the same computer hardware at the same time, which means that that calculations have to be scheduled, and thus resulting in delays. It is also possible that the hardware is improved and thus executing calculations faster. This is often the case in software that runs on personal computers. Over time, new technology results in improved hardware, such as computer chips, which can execute the software faster than it was originally designed and tested for. Faster execution is not always an advantage as some systems require precise timing and faster execution has a big impact.

2.3.3 Changes in internal structure

A different type of robustness compared to the input and environment is robustness of systems when they are extended with new functionality, as given in Definition **R13**. An example is upgrading of internet protocols. New features must be implemented without interrupting functionality. Software engineers refer to this principle as 'online management' [17].
New functionality can be part of the internal structure but it may also lead to new input or output. In the example of vending machines, additional input, such as a new type of coins, should not effect the behaviour of the system when a customer inserts the old type of coins. A more drastic change in vending machines is, for example, new type of payments, like bank cards. A new hardware element may have to be build into the machine to accept new types of payment. Again this new type of payment should not affect the old, coined, payments. Changes in the internal structure are by Definition 2 not considered in the environment. Nevertheless, the effects of changes in the internal structure lead to the same effects as with an unreliable environment. Furthermore, extensions to the system may include new input. New input that is added in the future, is input outside the domain in the present. This new input however, may go deeper into the internal structure of the system, whereas input outside the domain is refused by the system in an early phase. In the case of a vending machine, assume that a new button **D** is added that doubles the output of the machine. If the customer buys a chocolate, and presses **D**, he receives 2 chocolates. If we test the original machine, without **D**, on input outside the domain, the input is refused in the panel component. In the new system, the action of the new button passes the panel component and is passed around inside the system. The component that handles the communication between the panel and the output component (the mechanical component that picks the correct product), now is involved whereas in the case of testing outside the domain, it was not. In short, testing against new functionality includes testing internal components on invalid input. If we recall the effects of an unreliable environment, we also listed invalid internal data.

If we consider testing, we do not need an additional category for new functionality. For unreliable environments, we have given a set of effects that are the result of an unreliable environment. These effects are also applicable to new functionality.

2.4 System robustness categorization for testing

This section defines a set of categories that can be used to categorize testing of robustness in system engineering. To validate the completeness of the categorization the quality characteristics, given in Section 2.2.1 are used.

To make the categories practically usable, we want to generalize as much as we can. If we consider the characteristics in invalid input, we have already explained that some characteristics do not require a different approach in testing. Testing observable or direct input can be generalized by making observable inputs direct. The same holds for continuous and event-based input. Continuous input can be converted to event-based input. In the case of data and non-data, we have explained that invalid non-data input has no effect on the software, but only on the mechanical parts. Failures or changes in mechanical parts are considered in the environment and internal structure. For testing we therefore do not need to distinguish input in data and non-data. What remains is the content of the input. As explained, valid content of input, but invalid timing, requires a different approach in testing than testing inputs outside the domain. In addition malicious input requires a different approach than input out of domain in general. This means that within the input category, we have three separate categories, invalid timing, out of domain, and malicious input. Within these categories input is considered to be data, direct, and event-based.

For the environment, we have given a number of effects of an unreliable environment. One of the effects considers altering of input of the system. As explained, the effects on input, caused by the environment, are covered in the input categories. The remaining effects are considered to be different categories.

Effects of changes in the internal structure are essentially the same as the effects of an unreliable environment. We do not need additional categories in testing the robustness of the internal structure.

We end up with eight categories, which are listed in Table 2.1. To validate the completeness of the categories, we use the characteristics that are given in Section 2.2.1. Table 2.1 shows how the different characteristics are considered in the categories. All the characteristics are covered by at least one category. This does not mean that all robustness is covered, but it does provide some idea of the completeness of the categorization. What is notable is that recoverability is considered in all the categories. Recoverability is defined as the degree to which a system can recover in the event of an interruption or failure. These effects can be explained as, changes in the environment

Category	**Quality characteristics**
Inopportune or lack of input	Availability, Recoverability
Input out of domain of the system input	Recoverability
Malicious input	Recoverability
Loss of elements	Availability, Recoverability, Modifiability
Interference in behaviour	Fault tolerance, Recoverability
Corrupt, lost, or unexpected (internal) data	Co-existence, Recoverability
Corrupt, lost, or delayed communication	Co-existence, Recoverability, Adaptability
Loss or gain of time	Co-existence, Recoverability, Modularity, Modifiability, Adaptability

Table 2.1: Categorization in robustness testing and corresponding quality characteristics

or input, in terms of robustness. In order to be robust, the system should recover to acceptable behaviour.
An important part of deciding the importance of tests, is risk analysis. Test cases are often decided on risk analysis. Less frequent failures with higher risk might be more important to test than frequent failures with low risk. In [6] a framework is provided for assessing the risk of a damaged system. This method uses different probabilities of exposures that lead to, or contribute to, potential damages in the system. Risks are calculated, using the probability of exposures, the probability of the exposure leading to damage, and the probability of damage leading to a failure in the system. This leads to a risk analysis based on different failures of the system. In the proposed categorization in this thesis, categories are formed by means of consequences to the system behaviour. These consequences can be seen as failures in the system and therefore the method of risk analysis proposed in [6], can be used with the proposed categorization to asses risk.

3 A Model-based approach to robustness

In this chapter, a model-based approach is proposed to test the robustness categories that are proposed in Section 2.4. This thesis considers two model-based techniques, model checking and model-based testing. With the current tools available, these approaches can, separately or in combination, be applied in practice. Model checking provides mathematical proof that specific behaviour is not possible in the model. On the other hand, model-based testing provides a verdict whether the system stays within the behaviour that is specified in the model.
This chapter provides background information of the fundamental concepts of model-based testing and model checking. Furthermore methods are provided to test the different categories in robustness. The described methods in this chapter will be used to analyse the robustness of lighting systems in Chapter 5.

3.1 Labelled Transition Systems

In this thesis we consider models to be (timed) *Labelled Transition System* (LTS) and *Input Output Transition Systems* (IOTS).
A LTS S is formally defined as a quadruple (Q, q_0, L, T_S) where Q is a finite set of states, $q_0 \in Q$ is the initial state, L is the set of actions, and $T_S \subseteq Q \times L_\tau \times Q$, where $L_\tau = (L \bigcup \{\tau\})$ where τ represent internal (unobservable) transitions, is the transition relation [31].
A sequence $\sigma \in L*$ is called a *trace* of S from state $q_1 \in Q$ if there exists a path $(q_1, a_1, q_2)(q_2, a_2, q_3)...(q_n, a_n, q_{n+1})$, such that $\sigma = a_1...a_n$.
An IOTS is a LTS where input and output are labelled actions and input is never refused. An IOTS is formally defined as a quintuple (Q, q_0, L_I, L_O, T_S) where L_I and L_O are disjoint sets of input and output actions, respectively [28, 31]. In addition to a LTS, an IOTS is input enabled and therefore weaker than an LTS because an IOTS never refuses an input whereas a LTS does, $IOTS(L_I, L_O) \subseteq LTS(L_I \bigcup L_O)$.
An example of a graphical representation of a IOTS is shown in Figure 3.1. Input and output are distinguished by question marks and exclamation marks, respectively. In the example, $L_I = \{event?\}$ and $L_O = \{action!\}$. In addition, IOTS may include internal transitions τ.

3.1.1 Time

In the case study that has been analysed, time plays a critical role. When time is added to the specification, input and output are restricted to time. Timed Input Output

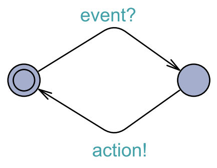


Figure 3.1: Example of a simple input/output transition system in Uppaal

Transition Systems (TIOTS) are labelled transition systems extended with time. Labels are either an observable action (input or output), a time passage action, or the internal action τ [31]. A TIOTS is formally defined as a sextuple $(Q, q_0, L_I, L_O, D, T_A)$ where D is the set of time passage actions and $T_A \subseteq Q \times (L_\tau \bigcup D) \times Q$. In addition to IOTS, letting time pass moves a TIOTS to a new state.

3.2 Model checking

Model checking is a reasoning technique to validate whether mathematical properties hold. In model checking, a model describes the behaviour of the system. Properties define the allowed behaviour of the system. Typically, properties indicate whether particular states are reachable. A property can for instance be 'The system should never reach location **B**' or 'The variable **t** should never be larger than 500' in the case states contain discrete variables. The proces of model checking is shown in Figure 3.2. A model checking tool has input in the form of a model **A** that describes the behaviour of the system and a set of properties **F** that defines the allowed behaviour. A model checking tool, algorithmically checks whether a property holds in the model and outputs the verdict.

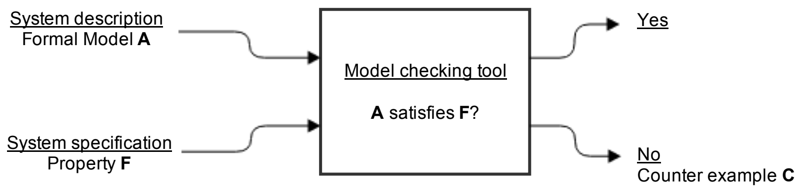
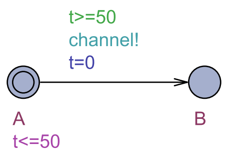
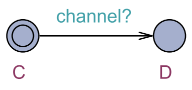


Figure 3.2: Model checking

In model checking, properties are defined in a temporal logic language [10]. This thesis considers the model checking tool Uppaal. The language that defines properties is explained in Section 3.2.1.
In model checking, it is possible to mathematically prove that a system does not reach a state that is not acceptable. For example, in the case of a vending machine, mathematical proof can be given that the machine does not output a product without first receiving the necessary amount of coins. This does however require that the vending machine is



(a)



(b)

Figure 3.3: Example of Uppaal model

a correct implementation of the model that is used in model checking.

3.2.1 Uppaal

The model checking tool that is used in this thesis is Uppaal. This section gives a brief description of how models are structured in Uppaal. It does not cover the whole modelling language but only the parts that are required to understand the models in the remainder of this thesis. A full detailed description of the Uppaal modelling language can be found in the Uppaal tutorial [7]. Furthermore this section explains how properties are defined in Uppaal. A number of examples is provided to show how Uppaal can be used for model checking.

Timed automata

A model in Uppaal is a network of timed automata. Automata in Uppaal are finite state machines extended with clock variables. A state in Uppaal consists of a location, values of clocks, and discrete variables. As an example consider the model in Figure 3.3. The model consists of two automata. The automaton in Figure 3.3a has two locations, **A** and **B**. The initial location **A** is highlighted with a double circle.
Locations are connected by means of edges which are displayed as arrows. An edge can be triggered and will lead to a new state. Edges can be labelled by guards. In the automaton in Figure 3.3a the guard **t>=50** indicates that the transition can not occur when the value of clock **t** is smaller than 50 time units. Furthermore, edges may include update statements. In the example, **t=0** resets clock **t**. Locations in Uppaal can include invariants. The system can only be in a location if the invariant holds. In the example, location **A** has an invariant **t<=50**. The system is only allowed to stay in location **A** when clock **t** is smaller or equal to 50 time units. Guards, updates, and invariants can include operations on clocks or operations on discrete variables.
To connect different automata in the network of automata, Uppaal makes use of, so called channels. A channel consists of an action, noted with an exclamation mark (**channel!** in Figure 3.3a), and, so called, co-actions, noted with a question mark (**channel?** in Figure 3.3b). Whenever an action occurs, a co-action must follow. In the example this means that when the edge between **A** and **B** is triggered, the edge between **C** and **D** is also triggered at the same time. An action can take place only when a co-action can take place. If there are more than one co-action that can take place, the behaviour

is non-deterministic and only one co-action will follow. A special kind of channels are broadcast channels. Actions in broadcast channels can always take place. When a broadcast channel is triggered, all co-actions that can follow, will follow.

Model checking with Uppaal

In Uppaal, properties are defined with the properties, listed in Table 3.1, where **p** and **q** define what expression should hold [7]. Expressions apply to automaton locations, discrete variables, clocks or constant values, and can be expressed with binary operators such as **and**, **or**, and >=.

Name	**Property**
Possibly	E<> p
Invariantly	A[ ] p
Potentially always	E[ ] p
Eventually	A<> p
Leads to	p −− > q

Table 3.1: Property expressions in Uppaal

Not all expressions are explained, instead some examples are provided to illustrate how properties in Uppaal can be defined. A full description of the Uppaal property language can be found in the Uppaal tutorial [7]. Examples of Uppaal properties:

A[] t>5 and t2 <= 3
: True when in all states, **t** is greater than 5 and **t2** is smaller or equal to 3.

E<>sample.A
: True when location **A** in automaton **sample** can be reached. **sample** in this case, is a automaton in the network of automata.

A[] not deadlock
: True when all states are not dead-locked.

3.3 Model-based testing

Testing is an important technique to increase confidence in the quality of a (computer) system [31]. The goal of software testing is to apply test suites for a software product under test to validate the quality of the software. Test suites consist of a number of test case specifications. Test cases are generated from the specification of the system. Based on the executions of the test, a verdict of the correctness of the IUT (implementation under test) is given [31].
In model-based testing, a formal model represents the behaviour of the IUT in the same way as with model checking. With the use of appropriate tools, model-based testing is performed fully automated, once a test model is available. This means that, in

comparison to manual test generation, tests are not human-biased. Instead an algorithm decides what behaviour is tested. This means that behaviour which is not considered to be 'problematic' in the eyes of the implementer, is also tested. Another advantage of automated test generation is that, when the specification changes, the model can be altered and the process of test generation can be repeated without the need of manually altering the test suite. Furthermore, the model in model-based testing is unambiguous and therefore there is no room for misinterpretation between the person who implements the system and the person who tests the system.
Model-based testing is considered black-box testing as the IUT provides a predefined set of inputs and outputs for the tester to interact with the system. Nothing is known about the internal implementation details during the test execution [33].
The process of model-based testing is shown in Figure 3.4. A model-based testing tool receives input in the form of a formal model that describes the behaviour of the system. By means of providing input to the IUT and observing its outputs, the tool provides a verdict whether the IUT is a valid implementation of the model. Details considering the verdict of the model-based testing tool are provided in the remainder of this section.

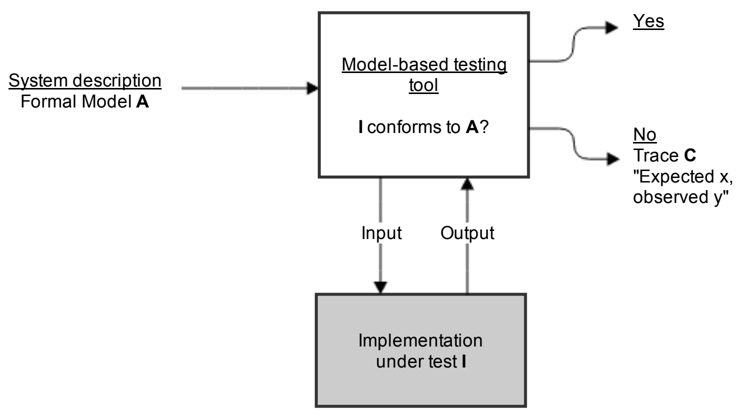


Figure 3.4: Model-based testing

3.3.1 Input Output Conformance

In order to make the verdict whether an implementation conforms to its specification, a notion of conformance is needed. Ideally we would like to have a test suite that is complete and therefore can distinguish between conforming and non-conforming implementations [31]. In practice this is difficult because complete test suites are usually infinite. This gives the need for a weaker conformance relation. The conformance relation should be sound; all correct implementations must pass the test. However, possibly some incorrect implementations will pass the test as well.
The notion of input output conformance (ioco) is a widely accepted conformance relation in model-based testing. Informally, an implementation i is ioco to a specification s if

the output of each trace of s, performed in i, results in a subset of outputs of the same trace, executed in s. The formal definition is as followed [31]:

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

where

- $Straces(s)$ are the suspension traces of specification s. Suspension traces consist of input and output, as well as, quiescence δ. Quiescence transitions are transitions between states where no output is observed.
- $s \ after \ \sigma$ is the set of states in which s can be after having executed trace σ.
- $out\,(i \ after \ \sigma)$ is the set of output which may occur in some state of $i \ after \ \sigma$. Possible output includes no output, quiescence δ.

3.3.2 Relativized Timed Conformance

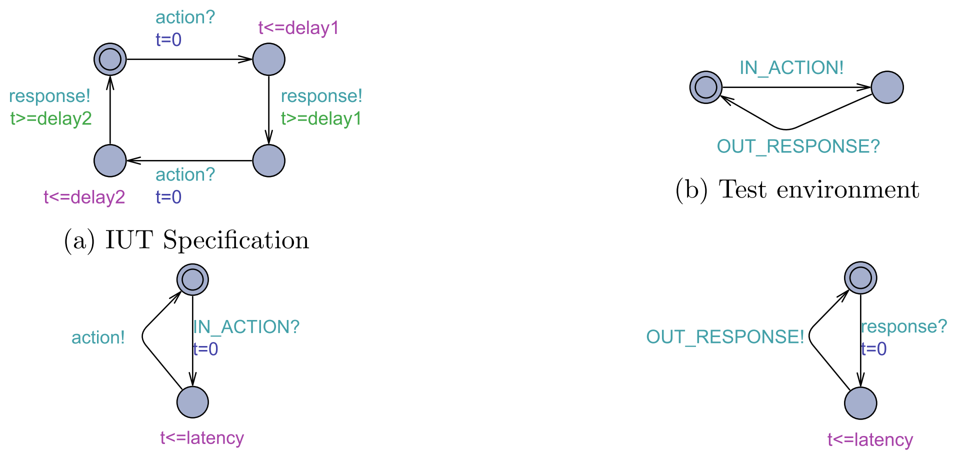
Different extensions of ioco are defined for time [26]. In this thesis we consider rtioco, as defined by [19]. Rtioco is the conformance relation used by Uppaal Tron, which is used in the experiments performed for the lighting system. The main feature of rtioco is that the environment of the IUT is considered explicitly. The conformance relation is shown under such an environment. An implementation i is $rtioco_e$ with specification s, in environment e, when executing any timed input/output trace σ, that is possible in the composition of e and s, the implementation i in environment e must only produce outputs and timed delays which are included in the specification s under environment e [19]. The formal definition of rtioco, we take from [26]:

$i \ rtioco_e \ s \ \Leftrightarrow \ _{def} \forall \, \sigma \ \in \ nttraces\,(e) : \ out_{aa}\,((i, e) \ after_t \, \sigma\,) \subseteq \ out_t\,((s, e) \ after_t \, \sigma\,)$
where

- out_t extends out in ioco with permissible delays that the system can produce.
- $nttraces(e)$, normalized timed traces of e, are $Straces$ that include time delays. Time delays are normalized in the sense that a trace does not contain consecutive delays but just one, longer, delay instead.

3.3.3 Uppaal tron

In the experiments in this thesis, we make use of the tool Uppaal Tron. Tron is a timed model-based testing tool that makes use of Uppaal models and provides a verdict whether the IUT is rtioco to the model. In comparison to model checking, the models in Tron consist of two parts. In addition to the description of the system, the model contains a test environment. The test environment defines what input and output the system expects. We consider the example in Figure 3.5. In this example we have an IUT that produces the output **response** after receiving an input **event**. To make it interesting, the first time it receives **event**, **response** is produced after **delay1** time units. The



(a) IUT Specification

(b) Test environment

(c) Adapter

(d) Adapter

Figure 3.5: Example of Uppaal Tron model

second time it receives **event**, **response** is produced after **delay2** time units. The test environment in Figure 3.5b describes the allowed input and output behaviour of the system. In this example, the system can receive the input **IN_ACTION** and observe the output **OUT_RESPONSE** afterwards. Tron uses environment models to decide which inputs can be produced and which outputs can be observed when the system is in a particular state. Tron randomly chooses between producing an input or letting time pass and observe the IUT [19]. To connect the test environment with the IUT, adapter automata, shown in Figure 3.5c and 3.5d, are used. Adapter automata may include delays between the tester and the IUT. This ensures that the test does not fail due to delays in the communication between the test tool and the IUT that results in observing the output too late. In our experiments, Tron is connected to the IUT by means of TCP. The **latency** in the example could, for instance, represent the delay in TCP communication.

3.4 Model checking combined with model-based testing

The preferred approach in model driven system development is to construct and validate a model in the design phase of the V-model, before starting the implementation phase. After an implementation is finished, model-based testing should be used to validate whether the implementation is a valid implementation of the model. Figure 3.6 displays the process. With a model checking tool, the model of the system should be checked with properties that define the allowed behaviour. If the model passes the model checking phase, the implementation is build. If the model does not pass, the model should be repaired. After building the implementation, the model and the implementation are tested on conformance with model-based testing. If the implementation passes the test, we have an implementation that is a valid implementation of the model and therefore a

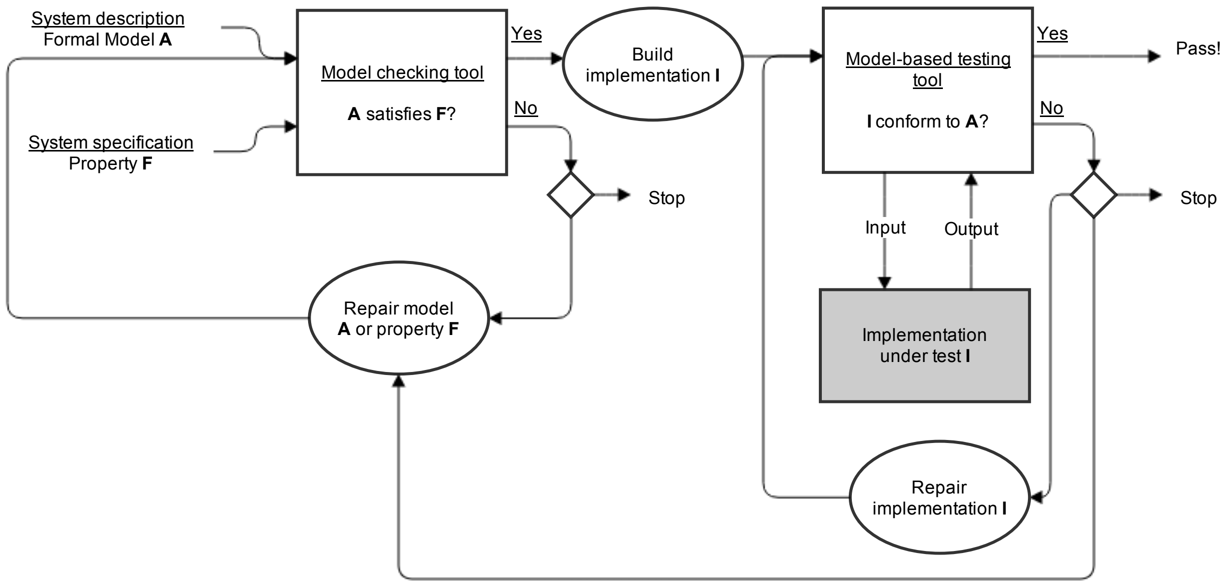


Figure 3.6: Model-based system development

valid implementation of the specification. If the implementation does not pass the test, then either the implementation or the model has te be repaired and the process should be repeated. The use of model checking will detect failures in the design and thus in an early phase of the development before the implementation is build.

3.5 Model-based robustness analysis

In the previous chapter, a categorization of robustness in system engineering was proposed. In this section, these different categories are used to explore how a model-based approach can be used to analyse and test robustness of systems. We describe each proposed category individually.

3.5.1 Inopportune or lack of input

Inopportune input, is input within the domain of the system input, provided to the system at unexpected timing or order.
To formally define inopportune input in terms of an IOTS, I first define input inside the domain. Input inside the domain of the system can only be defined with respect to a specification. The specification defines what is acceptable input. I consider the specification to be an LTS as defined in Section 3.1 with input and output actions L_I and L_O respectively. The specification is not input enabled. The input of a system, according to Definition 3, is considered to be all possible input of the system. When considering robustness, input can be anything, including input that the system was not

designed for. The size of all possible inputs is therefore enormous. Input inside the domain, however, is a subset of all the input and can be very small, for example, a software system that only accepts a boolean as input has only 2 possible inputs.
I define input within the domain of the system, to be all the input in specification S and thus L_I of S. In addition L_I is the set of all input in all states, such that

$$L_I = \bigcup_{q \in Q} L_{I,q}$$

where

- $L_{I,q} = \{a \in L_I | \exists q' \in Q : q \xrightarrow{a} q'\}$ is the accepted input in state q.
- $q \xrightarrow{a} q'$ defines that there exists a transition with action a from state q to state q'.
- Q is the set of states of S.

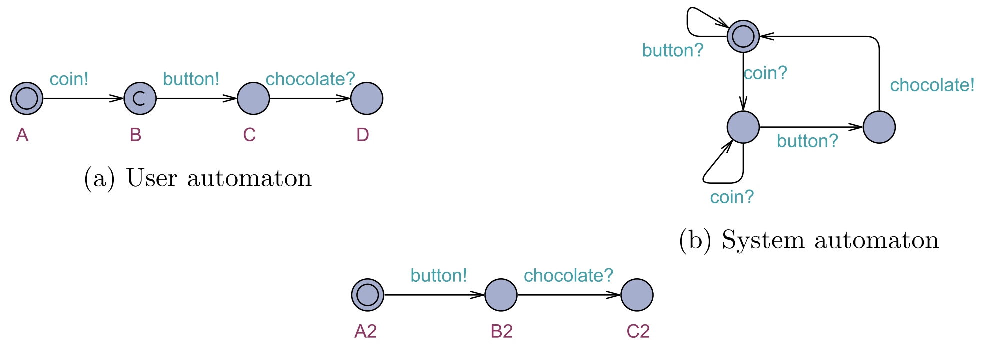
If a system I is considered an IOTS, then $L_{Idom}(I) = L_I(S)$ is the input within the domain, where S is the specification of I. Furthermore $L_{Idom,q}(I) = L_{I,q'}(S)$ is the input within the domain in state q where q' is the corresponding state in S.
Inopportune input L_{inop} can only be defined with respect to a state in the IOTS. Inopportune input in one state, is valid input in at least one of the other states.

$$L_{inop} = \{(L_{inop,q}, q) | q \in Q\}$$

where

$$L_{inop,q} = L_{Idom} \setminus L_{Idom,q}$$

The ioco relation in model-based testing, where i *ioco* s, considers all Straces in s. If s is input enabled, the Straces of s include all possible order of inputs because s accepts all input in all states. This means that there is no such thing as inopportune input because s accepts all input at all times. However, a specification is not always input enabled. Consider σ' to be a trace with inopportune input a_0,..,a_n such that $\exists x \in [0, n] | a_x \in L_{inop,q'}$ where $q' \in s$ *after* $(a_0, .., a_{x-1})$. This trace is not necessarily a trace of s and therefore s *after* σ' is not always defined. This means that, when considering the ioco relation, we cannot simply alter the Straces, to include inopportune input or remove inputs, and compare the result when performed in s and i. What can be done, is include the inopportune input in s to make it either partly or completely input enabled. The result is that now the inopportune input is part of the Straces of s and thus considered in ioco. The same holds for removing inputs, removing an input transition results in the input being absent in the Straces.
In model checking, the model is also not input enabled and thus, executing a trace with inopportune input, is not possible. In model checking the model also has to be altered to enable inopportune, or disable allowed input. Typically a model explicitly defines the input of the system. For example, in the case of a vending machine, the model in



(a) User automaton

(b) System automaton

(c) Alternative user automaton

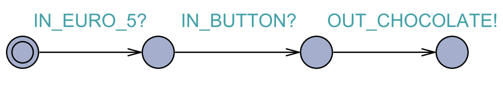
Figure 3.7: Example of vending machine

model checking typically includes the interaction of the user. An example in Uppaal is given in Figure 3.7a. Here the user inserts a coin, then, immediately presses the button and then receives chocolate. This behaviour is considered to be the normal behaviour. The system, in Figure 3.7b describes the behaviour of the system when it receives coins and button presses. To analyse the behaviour of unexpected timing of input, we can create a different model of the user to test the response of the system. Figure 3.7c provides an alternative user model. Here the user only presses the button and expects to receive chocolate. If we specify that the machine should not output chocolate without observing a coin, **C2** should not be reachable. We can create a property in Uppaal with the expression 'A[] not C2'. This will check that it is never possible to be in location **C2**. The same approach can be used to test unexpected timing of input. We could create a user model where the button is pressed, for example, 30 time units after the coin is inserted, by inserting a guard in the transition between **B** and **C**.
In model-based testing, the model also defines the input of the system in a similar way, and thus the same approach from model checking can be used. In the case of Uppaal Tron, where a test environment model is used, we can use different test environment models which can include delayed inputs.

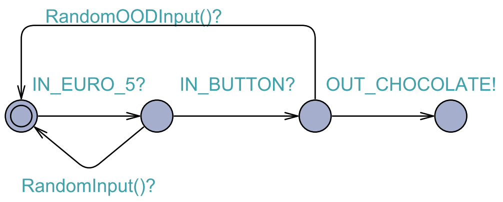
3.5.2 Input out of domain of the system input

Input out of domain is all the input of a system that was not considered in the design of the system. Formally, input out of domain can be defined as $L_{iood} = L_I \backslash L_{Idom}$ where L_{Idom} is input within the domain of the input of the specification of the system, defined in the previous section.
To test the behaviour of a system after it received input outside the domain, a method is needed to generate input. Model checking cannot be used to check against unexpected input. Model checking considers a closed model where inputs and outputs are considered explicitly in the model. Creating new inputs will not affect the model behaviour. In the real system however, unexpected inputs can effect the behaviour of the system. This means that model-based testing is of interest.

What can be done is use a form of randomization to produce (pseudo-)random inputs. This can work for all data input. Creating this random input is something where model-based testing cannot be used. However testing whether the system remains functional when such a random input occurs is something where model-based testing is useful.
The added value of using model-based testing, is that invalid inputs can be inserted during the operation under normal behaviour. If we take the example of the vending machine in Figure 3.8a, we can first insert **Euro_5**, as expected, and then insert **Dollar_5** to test invalid input. The model should define what happens after unexpected input. This will test whether the system remains functional if something goes wrong halfway through the process. In a small case like the vending machine this can also be tested manually. However if the system becomes more complex and contains more non-determinism, constructing tests automatically is preferred. If the tester follows a model of the system, it can easily construct traces that contain invalid input at any point. Formally, a trace σ' contains invalid input when $\exists a \in \sigma' \mid a \in L_{iood}$.



(a) Normal behaviour



(b) Inserting random inputs

Figure 3.8: Sample of vending machine

There are two approaches that come to mind to insert invalid input in model-based testing. One possible method is to let the test tool randomly insert invalid input during the tests. The IUT should then remain functional and continue the test. This gives the same problem as with inopportune input, the model does not include a transition of the random input because it is not input enabled. In some systems the system must return to the initial state while in other systems the invalid input should just be ignored or an error message should be shown.
A second method, that does allow this, is to have input in the form of a computational function. An example is given in Figure 3.8b. The function **RandomOODInput** will produce a random input a' for the IUT such that $a' \in Liood$. The model specifies how the IUT should response to the invalid input by means of going back to the initial state. In the tools that are currently available, this is not supported. In most test setups, however, an adapter is used to connect the test tool with the IUT. This test adapter can be used to convert input from the tester to random input. For example, the tester produces the input **IN_RANDOM** and the adapter changes it to a random input.

3.5.3 Malicious input

Malicious input is a more special type of input outside the domain. Malicious input is difficult to formalize. As mentioned, malicious input is not part of the domain of the

input of the system. Malicious input differs in the sense that, even though all input outside the domain should be rejected by the system, the probability of a failure after malicious input is greater than after non-malicious input.
As with input outside the domain, model checking cannot be used to test behaviour after malicious input. For model-based testing, the same advantage holds as with input out of domain. We can insert malicious input much easier halfway through the process because we have a model that specifies when input and output is expected from the system. For malicious input, we can extend the method with inserting random input. Instead of inserting random input, we now insert malicious input. Because malicious input can be different in different states of the system, the model becomes more complex and requires that for each state malicious input is defined.
If we recall the example of the vending machine in Figure 3.8a, we can first insert **Coin_5**, as expected, and then insert **Coin_400** to test malicious input. After we have received the chocolate, we can try different buttons to try to get the money back. To model automated insertion of malicious input, we can use transition with executable functions to create malicious input, as shown in Figure 3.9. To specify what malicious input should be constructed, different functions can be used. In the example in Figure 3.9, we first have malicious input in the form of coins, but after the chocolate is received we have malicious input in the form of button presses.

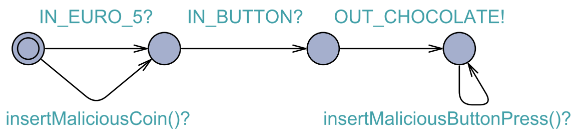


Figure 3.9: Inserting malicious inputs

3.5.4 Loss of elements

To model check robustness against loss of elements, we have to model the system in such a way that each element is modelled separately. This would allow removal of elements to test the behaviour when one of the elements is no longer available in the system. Because we can simulate the behaviour of the system when an element is no longer available, we do not have to physically alter the system under test. In model-based testing we would require to remove or deactivate elements in the real system. In practice this can be difficult. Consider an example of a vending machine with a heat sensor. If we want to validate the behaviour when the heat-sensor is temporary unavailable, we have to manually disconnect and reconnect the sensor from the machine. However an advantage of model-based testing is that we do not have to model elements explicitly. The model defines the allowed behaviour, if we alter the system and disable an element, the model still contains the allowed behaviour of the system. We could simple test whether the system remains conform to the model. In the ioco relation, i' $ioco$ s should hold where i' is an altered implementation where elements have been disabled.

3.5.5 Interference in behaviour

Testing robustness against interference in behaviour in general is difficult to test. There are many different outcomes when, for example, the source code of software is altered by environmental circumstances. What can be tested is the behaviour of components in the system when they receive unexpected input from other components. If we consider a model-based approach, model checking cannot be used but model-based testing can be used. In model checking, there is no implementation of the system but only an abstraction in the form of a model. Because we have an abstraction of the implementation, we cannot know what possible behaviour occurs when the behaviour of the system has been interfered with. Model checking is therefore not a good technique for testing robustness against interference in behaviour. In model-based testing, there is an implementation. The model in model-based testing, contains all valid behaviour of the system. We can test whether the implementation remains in valid behaviour in the model when faults are inserted in the implementation. This does however require some manual input for generating test. It has to be decided what kind of faults should be inserted in the implementation. This means that test generation is no longer completely automated.
To illustrate, we consider the example software and corresponding model in Figure 3.10.

```
a = 1 + 1
if a == 1:
        ouput TRUE
else:
        output FALSE
```

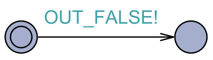


Figure 3.10: Sample software (left) and corresponding model (right)

The model defines that the system should always output **FALSE**. This means that if, for instance + in **a=1+1** is replaced with **-**, the system behaviour will still be valid according to the model. Because we have to decide what faults are inserted, we can say to what degree the system is robust because we know which faults will make the test fail.
There are two approaches possible. Either we let the tester decide when faults are inserted, as with inserting random input, or we let the environment insert faults. The advantage of letting the tester decide on fault insertion is that tests are much easier to re-execute as there is one point-of-decision in the test.

3.5.6 Corrupt, lost, or unexpected (internal) data

Loss of data requires a similar approach as with interference in behaviour. Instead of inserting faults in the implementation, we insert faults in the data, by either deleting or altering data. Altering of data in general is easier to achieve than altering the behaviour as it does not require to harm the implementation, whereas, for example, altering the source does. Again this method is only applicable to model-based testing and not in model checking.

3.5.7 Corrupt, lost, or delayed communication

Loss of communication is one of the things that can easily be included in the model of the system. To model internal communication, messages can be considered input and output of the system. This means that when testing internal communication, some structure is considered of the system and thus testing is less black box but more towards grey box.

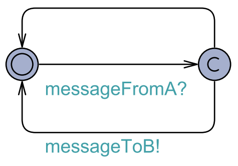


Figure 3.11: Communication automaton

In model checking, communication can be modelled as shown in Figure 3.11. The automaton observes a message, by means of a co-action **messageFromA**, and either forwards it, by means of an action **messageToB**, or drops it, by means of an empty transition back to the initial state. Model checking can then be performed to validate whether the model still holds the properties that are defined in the specification.
In model-based testing, there are two possible methods to test against message loss.

1. The tester decides on message loss: All messages go through the tester and the model. The model contains an environment that decides when messages are dropped or not.

2. The environment decides when a message is lost: The environment randomly drops messages. The tester has no knowledge whether a message is lost or not. The model should be non-deterministic to allow message loss.

Both methods have advantages and disadvantages.

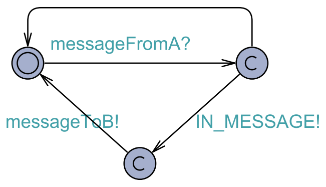


Figure 3.12: Communication environment automaton

Method 1

The first method lets the tester control the environment and makes the decision whether a message is dropped or not. The decision of a message drop can be sent to the IUT by means of input. The model contains an environment automaton which, upon receiving a message, decides whether the message is forwarded or not. Figure 3.12 shows an example of such an environment automaton. The capital channel, **IN_MESSAGE** is input for the IUT to send a message to the other element.
An advantage of letting the tester decide when messages are lost, is that there is only one point of decision making. This has the advantage that when a test fails, the test case, or trace of inputs, can be repeated when the IUT is modified. A disadvantage of this method is that, in order to let the tester control the environment, additional delays are introduced in the process. The tester should decide on message loss, as soon as, a message arrives at the tester.

Method 2

In the second approach, the environment is controlled by the adapter between the IUT and the MBT tool. The adapter decides when messages are dropped or not. The tester now only has control of the IUT, and thus the model should consider message loss whenever a message is sent internally in the model. This requires communication to be modelled that same way as with model checking, as shown in Figure 3.11.
An advantage of this method is that the model only contains the behaviour of the real IUT and does not control the environment in which it operates. A disadvantage is that the complexity of the model increases, as more non-determinism of the IUT is introduced. In addition, replicating the test becomes more difficult because the test trace does not include information when a message was lost. In the ioco relation, i' *ioco* s should hold where i' is an altered implementation where communication is disrupted.

3.5.8 Loss or gain of time

Testing the system with faster calculation is not always possible in the real system due to limitations in hardware. In general slower calculations are easier to test because it does not require faster (and often more expensive) hardware. With model checking, time is simulated. What can be done is insert time delays in the model to simulate longer or faster calculation time. The model should then still hold the properties from the specification.
In model-based testing, hardware can be physically changed or affected, for example, by executing heavy calculation simultaneously with the test execution. The implementation should remain within the behaviour that is specified in the model. In the ioco relation, i' *ioco* s should hold where i' is an altered implementation where calculation speed is affected.

4 Case study: Smart indoor lighting systems

This chapter describes the lighting system that has been studied to test robustness against message loss. An introduction to the system is provided alongside a basic description of how it is technically implemented. This chapter also explains which part and configuration of the system is considered in the experiments in Chapter 5. Furthermore the implementation of an automated test setup for the lighting system is explained in this chapter.

4.1 Smart indoor lighting system

The lighting system that is being studied is a smart lighting system that requires almost no direct interaction from the user. The system detects whether people are in the room and can adjust the brightness according to external light supply, such as daylight. There are no physical buttons connected to the system to switch the light on or off; the system decides when the lights are switched on or off.
Rooms are configured with different predefined light settings. Different settings are referred to as *presets* where, for instance, preset 1 is configured to provide the maximum amount of light and preset 2 is configured to produce no light. The system can then be configured such that the luminaires are by default in preset 2 and switch to preset 1 when occupancy is detected. Other predefined presets can for instance, be optimized for presentations, relaxing, or concentrating. Different settings can be applied via a remote control device or personal smart phone.
An additional feature of the lighting system is support for areas to be connected to a parent area. The lights in the parent area should be on when either one of the children or the parent itself detects occupancy. An example is the hallway in an office building that connects offices. The hallway should produce light for as long as at least one office is occupied.
In this thesis, the focus will be on lighting systems that are installed in large office buildings. The system provides a full lighting solution that ranges from office spaces, restrooms, corridors, but also larger areas, such as lounges, and entrance areas. The number of luminaires in such a building is in the order of thousands.

4.2 Technical implementation

The luminaires communicate with one another by means of IP (Internet Protocol). Physically the luminaires are connected via Ethernet cables that provide both communication and power (Power over Ethernet (PoE)). This allows luminaires to be connected to the network and the power supply by just one cable.
Apart from providing light the luminaires may contain sensors. The sensors that are present in this system are motion sensors, to detect movement, and light sensors, to measure total light intensity from both luminaires and natural light in the field of view of the sensor.
The luminaires in the system contain a controller unit that can receive and send messages to other luminaires. These controller units contain a timer that is used to decide when luminaires should no longer produce light. To avoid inconsistencies, timers are kept synchronized.
Luminaires are grouped by means of control groups, called *areas*. Luminaires within an area should be identical in behaviour. An area is typically a room or a corridor, but it can also be part of a room. Luminaires are physically connected to a PoE switch, which is in turn connected to a parent switch to form a hierarchical structure.
The system is highly configurable. There are many system configurations possible, not only in the number of luminaires and sensors, but also in the configuration of the presets, building topology, deployment, and network topology.

4.2.1 System and environment

To define what is considered to be the system and what is considered to be the environment, recall the definitions given in Chapter 2. A system consists of a number of elements that together produce a result that cannot be obtained by the elements alone. In the case of the lighting system, we have a set of luminaires that communicate with one another by means of Ethernet communication. Physically the communication goes via off-the-shelf PoE switches. In addition we have panels and software running on smart phones, that can change the preset of the luminaires. The latter two however, are not within the scope of this thesis and are not considered in the experiments.
We can apply the method described in Section 2.2 to define a system. In the lighting system, the result that is being studied is the preset state, or light density, of the luminaires. The inputs that are required to achieve this result are motion detection, light density (existing light density in the room), and power.
The elements that provide the light density result are the physical luminaires. These luminaires require communication with other luminaires, input in the form of motion detection and existing light density, software to control the behaviour after receiving messages, and power. The element that provides the communication between the luminaires, as well as power, is the PoE switch. The design of the PoE switch however is not in control and therefore the PoE switch is not part of the system. The design f the software of the luminaires is in control and therefore the software of the luminaire is part of the system. What remains is the motion and light input for the luminaire. The

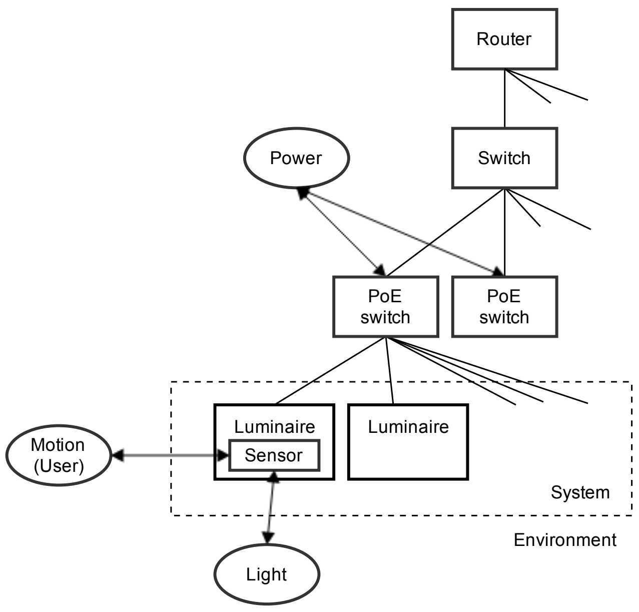


Figure 4.1: System and its environment

elements that provides these are the sensors or other luminaires. The sensor requires power, motion, and light. Power is obtained from the luminaire, motion and light are observed by the sensor. Motion, and light are required inputs to achieve the result and thus, the sensor is an entry point and the system is not further extended. All elements are now reached and the system is closed. The system consists of the luminaires, the software of the luminaires and the sensors.

The definition of environment states that every circumstance, condition, or element that surrounds the system, is part of the environment of the system. The size of the environment is very large, in the lighting system the environment includes users and external light but also room temperature and even the geographical location of the system. Not all elements in the environment are of interest. The room temperature for instance is not particularly interesting for the lighting system. External light is. Some luminaires contain sensors that measure the current light intensity in a room. The luminaires adjust the light output to provide the correct light density in the room. External light is not part of the system but part of the environment. The environment that is of interest in the lighting system that is analysed in this research consists of external light, the PoE switches, the network in which the PoE switches are connected, the users of the system that provide motion, and the power supply.

An overview of the system and its environment is given in Figure 4.1. This figure only contains the elements that are within the scope of this thesis.

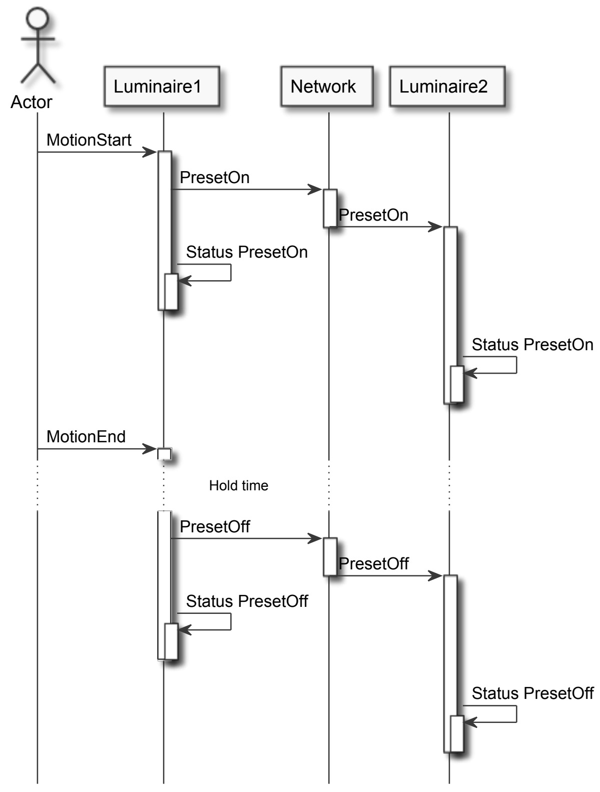


Figure 4.2: Specified behaviour after detecting motion

4.2.2 Motion sequence

The research in this thesis is limited to a basic configuration with one room and two luminaires, that both contain a motion sensor.
We consider two presets where one provides light and the other does not provide light. For convenience we call them *PresetOn* and *PresetOff* respectively. Figure 4.2 shows the specified behaviour of the luminaires under normal circumstances when occupancy is detected. When a person enters the room, a MotionStart event occurs and the room is occupied. **Luminaire1** detects motion and sends a PresetOn message over the network to other luminaires in the network, which in this case is only **Luminaire2**. **Luminaire2** receives the preset message and changes to the corresponding preset. When the person leaves the room, motion is no longer detected and a MotionEnd event occurs. Luminaires are configured with a certain *hold time* in which the system does not detect motion but is still in the PresetOn state. The hold time ensures that lights do not switch immediately off when no motion is detected for a short time. This avoids, for example, that the lights switch off immediately when the person in the room is not moving for one second, and then switch back on one second later, when the person moves again. When **Luminaire1** no longer detects motion, it waits until the hold time has expired and then sends a PresetOff message via the network to **Luminaire2**. **Luminaire2** receives the PresetOff and changes to the corresponding preset configuration.
All luminaires with a sensor have an internal clock and can send PresetOff messages. If a luminaire with a sensor receives a PresetOff from the network before the hold time expires, the luminaire will process the PresetOff, by going to the appropriate light level, and not send the PresetOff itself. Let us assume that only **Luminaire1** detects motion. It is possible that the behaviour is the same as in Figure 4.2, that is, **Luminaire1** sends PresetOn and later sends PresetOff. It is also possible that when **Luminaire1** detects motion, **Luminaire2** sends the PresetOff. However, it is also possible that both luminaires sent the PresetOff message. This will happen when the clocks of both luminaires are precisely synchronized.

4.2.3 Hold time synchronization

Let us now assume that both sensors detect motion. To simplify the situation, let us say that **Luminaire1** detects motion and **Luminaire2** detects motion one minute later, and both sensors stop detecting motion after two minutes. This situation is shown in Figure 4.3. When **Luminaire1** detects motion it will send a PresetOn to the other luminaires via the network. When **Luminaire2** detects motion, it does not send a PresetOn because it is already in the PresetOn state. After two minutes, **Luminaire1** does not detect motion and starts waiting until the hold time expires. One minute later, **Luminaire2** stops detecting motion. What is expected now is that **Luminaire1** sends a PresetOff, 1 minute before the hold time in **Luminaire2** expires, and thus switching the luminaires off 1 minute too soon. This results in an inconsistency between the lights which is not the preferred behaviour. To address this problem, luminaires with a sensor send synchronization messages during the time that the luminaire is in

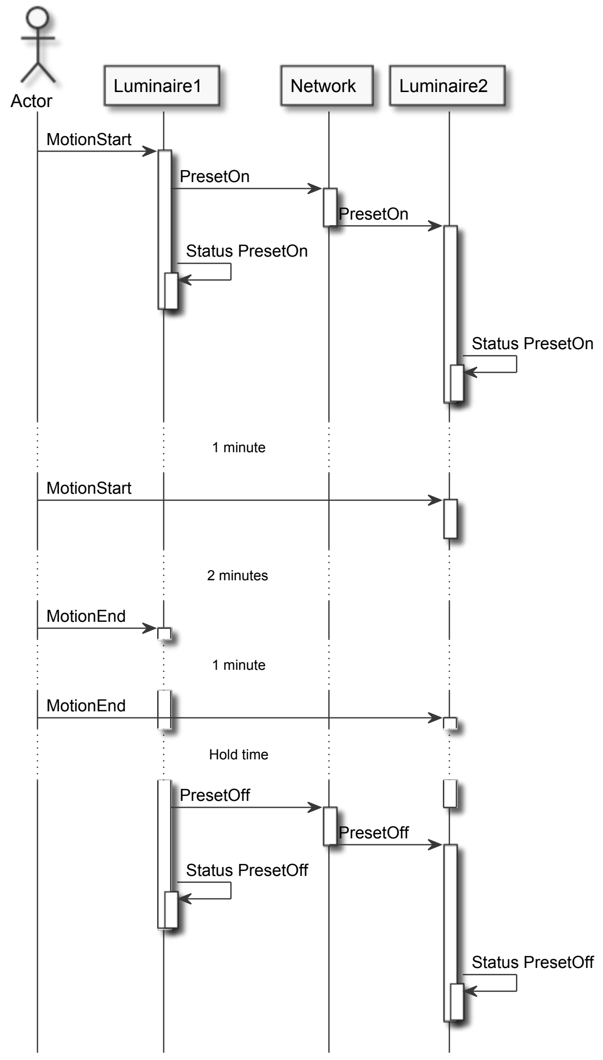


Figure 4.3: Specified behaviour when both luminaires detect motion

the PresetOn state. This synchronisation message ensures that the hold time in all luminaires is synchronized. As with PresetOff messages, it is theoretically possible that both luminaires produce the synchronization message precisely at the same time.

4.3 Robustness of lighting systems

The communication network used to connect the luminaires does not guarantee that messages are always delivered. This can be a problem when, for instance, a luminaire does not receive the message to switch to the PresetOn state, resulting in a situation where all luminaires in a room produce light, except for one.
Obviously when all communication is lost, the system will no longer be able to stay in a consistent state. Message loss of all messages is not a realistic situation. In practice even the probability of a single message loss is very low and depends on the amount of network traffic.
When we consider the robustness categorization in Section 2.4, we can place message loss in the category **Corrupt, lost, or delayed communication**. Because we only consider effects of communication loss, we do not have to distinguish between causes of message loss. As explained in Section 3.5, in a model-based approach to testing, the communication can be considered as inputs and outputs of the system, to control which messages are delivered normally and which messages are dropped or delayed. Details about the execution of the experiments and how robustness is modelled in the model, are given in Chapter 5.

Another factor of robustness in the lighting system, is power loss. The Dutch power network is rather stable. Outside the Netherlands, it is not always the case that the system runs on a stable power supply. The system should react in a proper way to a (short) power cut. This means that the start-up procedure is very important. When power loss happens, the system should restart and all the luminaires should produce the correct light level. Luminaires store the current preset periodically. This means that a luminaire might not have saved the latest preset when a power loss occurs, resulting in a different preset and light level of the luminaire when power returns. The question is not so much how often this appears, but rather whether the system recovers and if so, how long the system is in an inconsistent state.
Within the categorization proposed in Section 2.4, power failures are placed under the category **Invalid timing or lack of input**. The system expects to receive power continuously, however due to failures in the environment, power is interrupted. In testing, continuous input can be converted to event-based input, as shown in Section 2.3. In the case of power, one event can be triggered when the system starts receiving power and one event can be triggered when power is no longer provided to the system.

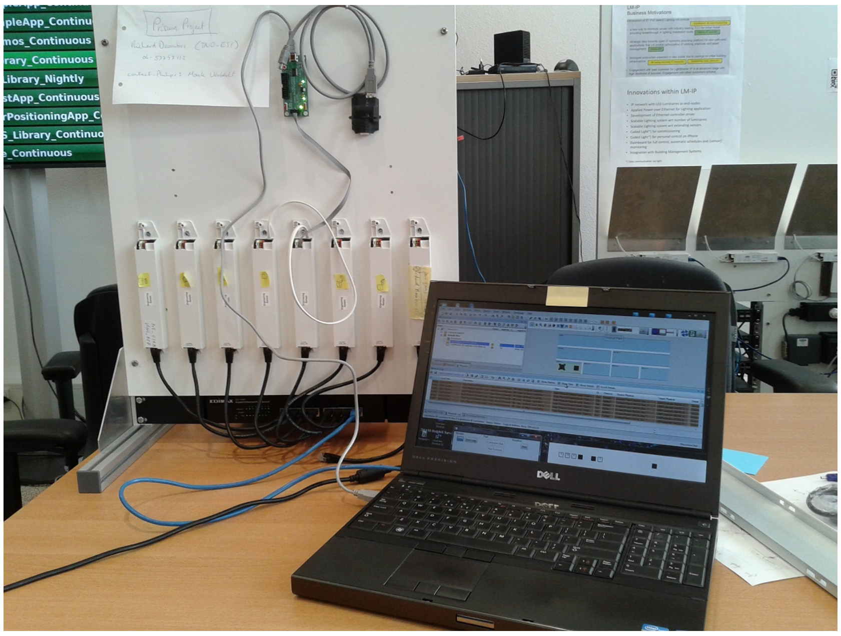
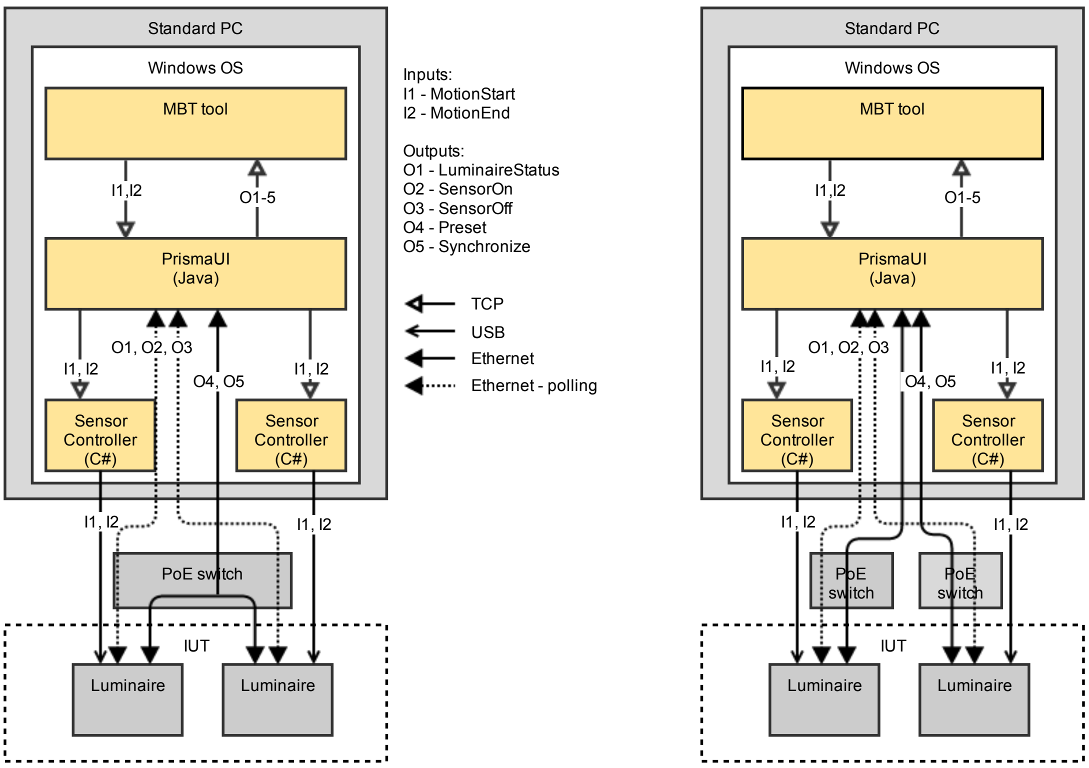


Figure 4.4: Test setup hardware

4.4 Test setup

The test setup that has been used to execute the tests contains a small number of luminaires. The physical hardware that is used in the tests, is shown in Figure 4.4. The hardware consists of six luminaires, two USB-controlled sensor simulators, and two PoE switches. In most of the experiments we have only used 2 luminaires. The reason for this limitation is that, when considering messages loss, luminaires have to be in a separate network. More luminaires would require more different networks. This is practically difficult because the PC that runs the test software, needs a network adapter for each network.
To test robustness against message loss, a second switch is used to create two separate networks that allow interception of network traffic.
Figure 4.5a gives a brief overview of the test environment architecture for testing normal behaviour, which means that both luminaires are connected to the same PoE switch. In addition, the data flow of input and output within the test setup is shown.
The luminaires communicate with each other via the PoE switch. The control of the input and output of the system is done via an adapter, so called *PrismaUI*. This adapter is connected to a sensor controller application that sends motion inputs to the luminaire via a USB-controller unit. The status of the luminaire is read via status-request messages over the PoE network (a laptop running the test software is connected to the PoE switch). The returned status contains information about the dim level of the luminaire and whether the luminaire's sensor is detecting motion.



(a) Testing in normal environment (b) Testing in controlled environment

Figure 4.5: Test architecture

In terms of test software, we have three components, the model-based testing tool, PrismaUI, and sensor controller software. A short description of each component is given below.

Model-based testing tool

A model-based testing tool algorithmically produces tests and provides input for the IUT (Implementation Under Test) and observes all output of the IUT. The MBT-tool communicates with PrismaUI to forward input to the real system. The construction of the software allows different tools to be connected. In the experiments only jTorX and Uppaal Tron were used. Results and details of the experiments are provided in Chapter 5. In both cases, TCP is used for communication between the tool and PrismaUI.

PrismaUI

The PrismaUI software provides a bridge between the MBT-tool and the IUT. PrismaUI observes the status of the IUT, that is, the light density of the luminaires and network communication between luminaires. In addition PrismaUI provides a graphical representation of the current status of the luminaires and the network traffic. Another important feature in PrismaUI is the manual insertion of inputs. We can for example, insert a MotionStart event during the testing. Doing so should make the test fail. This means that we can verify whether the test does fail with unexpected behaviour of the IUT.

Sensor controller

The sensor controller software does nothing more than receive MotionStart and MotionEnd events and forwards them to the physical hardware. Communication to the hardware goes via USB cables. Communication with PrismaUI goes via TCP.

4.4.1 Test environment with robustness

For testing robustness, we simulate message loss. This is done by using a second PoE switch and creating 2 separate networks. Each luminaire is connected to a different PoE switch. A laptop, running the test software, is connected to both the PoE switches and acts as a gateway for communication between the two luminaires. Figure 4.5b shows how the second PoE switch is integrated in the test architecture.

4.4.2 Motion sequence in test environment

In Section 4.2.2 the motion sequence has been explained when the luminaires operate in its normal conditions. In the test setup, the behaviour is the same but the sensor is now controlled by a piece of software. In addition the status of the luminaire is also

observed. The status of the luminaire in this case consists of the current level of light output and whether the sensor is detecting occupancy or not.
Figure 4.6 shows the (slightly simplified) sequence diagram in the test setup, synchronization messages are, for clarity reasons not shown. The **Tester** communicates with **PrismaUI** which acts as an adapter between the tester and the IUT. To control the sensor, **PrismaUI** communicates with software that controls the hardware that simulates the sensor. The IUT (luminaire) then receives the motion signal and sends out the **PresetOn** message. In addition the status change is sent to **PrismaUI**, which in turn forwards it to the tester. The **PresetOn** message that is sent to the luminaire is also received by **PrismaUI**, as it observes the network communication. **PrismaUI** notifies the tester that **PresetOn** was sent from **Luminaire1**.
The process for MotionEnd is similar except that the IUT waits until the hold time has expired before **PresetOff** is sent.

4.4.3 Order of outputs

The sequence in Figure 4.6 shows the specified order of outputs that the tester observes. In theory the first observation after a **MotionStart** is that the Occupancy sensor is detecting occupancy. The next output is a **PresetOn** message on the network. The final output after a **MotionStart** is a status change of the luminaire. In practice this is not always the case. The reason for this is that the messages that are observed from the network are real-time and the status change messages are not. Instead the IUT responds to status request messages. In the case of light intensity, the luminaire returns the light intensity of the luminaire upon request. However the status that is returned from the IUT is updated every 400 milliseconds. This means that we cannot observe the state of the luminaire in real-time which in turn, results in non-deterministic order of outputs. We expect to first receive the status change of the sensor but in reality it is often the case that the **PresetOn** message is observed before that. In the worst case, the occupancy sensor status change is received by the tester, 400 milliseconds after the **PresetOn** message was observed. The order of outputs adds non-determinism which makes it more difficult to model the behaviour. Furthermore, the behaviour in the test environment becomes different from the real behaviour because in the real behaviour the order of outputs is deterministic.

4.4.4 Latency

In an ideal world, the tester communicates with the IUT in real-time. Unfortunately additional software, including an operating system, is needed to execute the test software and adapter to the IUT. In the test architecture we can see that the test software depends on much additional software. Furthermore the communication between different components of the testing architecture, adds additional latency. The latency caused in the test setup has to be compensated in the timed specification in the model that will be used for model-based testing. Without compensating latency, the IUT will, according to the tester, never produce the output in time.

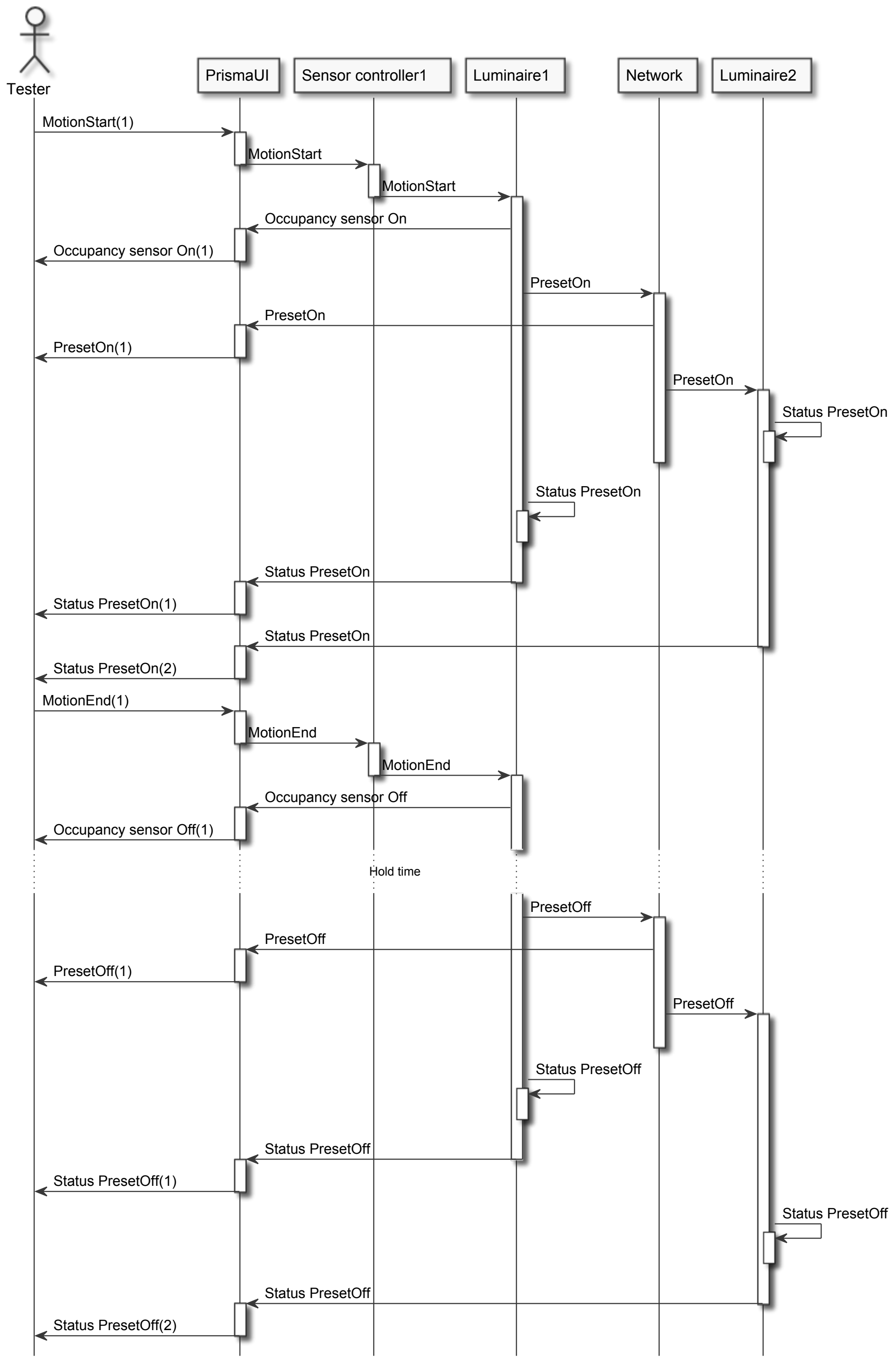


Figure 4.6: Behaviour in test environment

5 Model-based testing experiments

In order to test the robustness of lighting systems, the method described in Chapter 3 has been applied in a number of experiments. This section describes the experiments that have been performed, it includes how the model is constructed as well as the decision of MBT-tool that is used. To get going, the first experiments that have been performed, are limited to the normal behaviour under a trusted stable environment. From there the model, as well as the test setup, have been extended to test robustness against message loss.

5.1 Model-based testing tools

To decide what model-based testing tool to use, experiments have been performed with both timed and non-timed model-based testing tools; respectively, the tools Uppaal Tron [14] and jTorX [8] are used. Time is a critical element in the lighting system. In the tool comparison we found that it was difficult to abstract from time in the lighting system that is analysed. In the lighting system, outputs from the IUT are specified to be produced within a certain timespan. An example is sending the preset message that switches the lights off. This message should be sent after no motion is detected within the hold time. Abstracting from time means that this property cannot be tested because the allowed time span cannot be defined in the model. In addition we are interested in the period of time that the system is in an inconsistent state (e.g. one luminaire is off while the other luminaires in the same room are on). For these reasons a tool with time support is required.
A particular advantage of Uppaal Tron is that the existing Prisma models that have been created with Uppaal, can, in theory, be used with Tron. This is not only convenient but it also makes it easier to understand the models that are used for model-based testing, for the people involved in the Prisma project. For these reasons the tool Tron has been used for the remaining experiments. The tool Tron is explained in Section 3.3.3.

5.2 Model construction

Initially we intended to use the Prisma models for the model-based testing experiments. During the experiments we found that, in order to use the Prisma models, many changes had to be made to the model. Not only in the connection with Uppaal Tron, but also due to compatibility problems of the latest Uppaal version and the Uppaal version used by Tron. For these reasons, it has been decided to create a specialized model, based on the Prisma model, for model-based testing.

To make the Prisma model compatible for Tron, additional transformations have to be taken depending on features that are being used in the original model. The development of Tron is not up-to-date with the development of Uppaal, some of the new features in Uppaal that are useful for model checking, cannot be used with Tron. The Prisma model contains some of these new features and therefore cannot be used directly with Tron. One of the most important ones is the lack of copying clock values. Tron does not support to copy the value of one clock to another clock. This is a problem for the Prisma project as the luminaires do copy the value of the timers of other luminaires in order to synchronize the hold time timer. It is possible that this is resolved in a new version of Tron [22].
Another feature that is missing, is guards on receiving channels. This means that changes have to be made in the model to remove guards on receiving channels. This can be resolved by adding an additional committed state as shown in Figure 5.1. Note that this only works for broadcast channels. In normal channels, this would allow the transition with the action (channel! in the case of Figure 5.1) to take place. In other words, guards on receiving channels are also guards on sending channels. However in our models, it is applied to broadcast channels. In broadcast channels, the receiving channels will only follow when a transition with a channel is performed. In broadcast channels, guards in transitions with channels are only guards to the transition with a receiving channel.

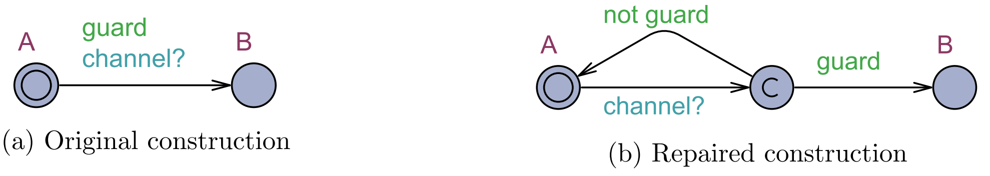


Figure 5.1: Receiving broadcast channels with guards

The input and output of the IUT are listed in Figure 5.2. The input is limited to motion, where motion is a period of time, which is represented as two events, MotionStart and MotionEnd to start and end motion respectively. The output consists of the state of the sensor (detecting motion or not), the (preset) state of the luminaire, and network traffic of preset and synchronization messages. All input and output is linked to a particular luminaire. The total amount of input for the IUT therefore depends on the number of luminaires in the test setup. In addition, some output is linked to presets. In the test we consider only PresetOn and PresetOff (recall Section 4.2.2).

5.2.1 Configurability of the system

The system that is being analysed is highly configurable. In an ideal world, all the different configurations should be tested. In reality this is infeasible and only a limited number of configurations can be tested. The system configuration can be divided in two main categories, the physical configuration and the parameterized configuration. The physical configuration consists, among other things, of the number of luminaires,

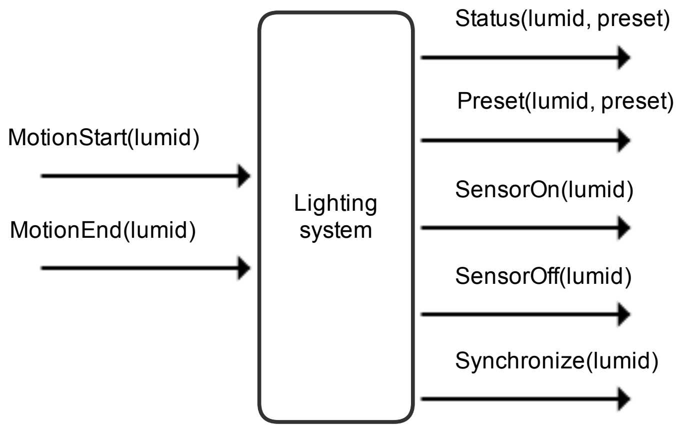
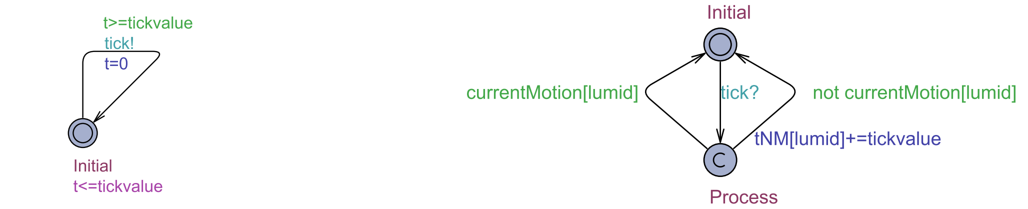


Figure 5.2: Input and output of the IUT

whether a luminaire contains a sensor, and in which area the luminaire is placed. Apart from the physical configuration, the system is configured with parameters such as the hold time and presets. The new model that has been created in this research allows different configurations of the system. Each luminaire consists of a couple of automata. The model can be generated depending on the number of luminaires and the presence of sensors. Different parameters, such as the hold time, are part of the model, by means of static variables.
The model is constructed in such a way that extensions in the system, such as corridor linking, can be added, by means of adding automaton. Messages between luminaires, for instance, are modelled with internal broadcast channels. Models that specify behaviour for corridor linking can observe these broadcast channels.

5.2.2 Copying clock values

The latest version of Tron does not support settings clocks to particular values. Only resetting the clock is supported. In the lighting system, copying clocks is a critical part of the behaviour of the system. Luminaires use synchronization messages to synchronize their hold time timer. This makes sure that luminaires stay in PresetOn for at least the hold time, after the last luminaire has detected motion. To model the synchronization messages, the clock value of the luminaire that sends the synchronization message, has to be copied to the clocks of the other luminaires. As Tron does not support copying of clocks, we have to work around it. To support clock copying, discrete clocks are used in the model. Discrete clocks are modelled by means of integer values that are increased periodically. This allows the copying of one clock value to another clock value. This does mean that the state set becomes much larger because every clock tick results in a new state.
The clock automaton in Uppaal is shown in Figure 5.3a. The clock automaton is a



(a) Clock automaton (b) Clock listener

Figure 5.3: Concrete clock in Uppaal

simple automaton that produces a tick, every *tickvalue* time units. A broadcast channel *tick* is used, to allow observation of the clock ticking. The clocks of each luminaire are modelled as integers, as shown in Figure 5.3b, that are increased when the tick channel is observed. In Figure 5.3b, *tNM* is an integer clock that holds the time ticks since the last motion was detected. In addition, *tNM* is not increased when the luminaire's sensor is still detecting motion.

5.3 Testing of normal behaviour

The first tests that have been performed, do not include message loss. What is tested in the normal behaviour is whether the luminaires produce preset messages at the right time and whether the state of the luminaires changes to the appropriate preset state whenever a preset message is received from one of the other luminaires. This section will describe the model that has been used in the test experiments. Not all the automata are discussed but only the most interesting ones.

5.3.1 Tester environment

Uppaal Tron uses environment models to define when the tester can produce input for the IUT, as explained in Section 3.3.3. The test environment of the lighting systems consists of a simple automaton for each luminaire as shown in Figure 5.4. The tester must observe that the IUT's sensor is detecting motion before ending the motion. The reason for this is that, as explained in Section 4.4, there tends to be a (random) delay between the tester producing the Motion and the simulated sensor actual seeing motion. The reason for observing the motion sensor status is, to make sure that the model can measure the time between the luminaire detecting motion and producing the PresetOff message, rather than measure the time between the tester starting motion and the luminaire producing the PresetOff message, which would include the random delay.

5.3.2 Test sensor

The model of the sensor is shown in Figure 5.5a. The model states that the IUT should produce the outputs indicating that the sensor is detecting motion and stopped detecting

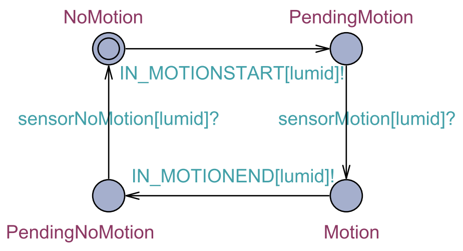
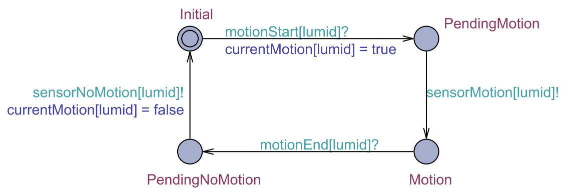
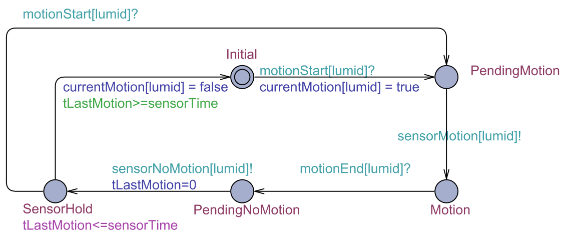


Figure 5.4: Environment automaton



(a) Real sensor



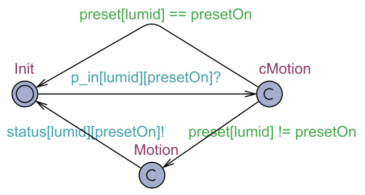
(b) Simulated sensor

Figure 5.5: Sensor automaton

motion when the tester produced the inputs MotionStart and MotionEnd.
To test the IUT, a sensor simulator, which is controlled by software rather than real motion, is used. The simulated sensor, stays in a motion detecting state for some time after the motion ends. However PrismaUI observes when the sensor stops detecting motion and this does not include the extra time.
To simulate the behaviour of the simulated sensor, the model in Figure 5.5b is used. This model has an additional state where the sensor keeps detecting motion and waits for *SensorTime*, which is a constant value set to the delay time, before going back to the initial state (unless new motion is detected).

5.3.3 Luminaire status

Upon receiving preset messages, the luminaires must go to the corresponding preset. Changes in preset are considered an output of the IUT. In the case of the PresetOn message, the IUT must produce the output that the luminaire status has changed to PresetOn unless, as shown in Figure 5.6a, the luminaire is already in the correct preset. When the luminaire receives a presetOff message while the luminaire's sensor is still detecting motion, than the luminaire should reproduce the PresetOn message to restore the other luminaires to the PresetOn state. Figure 5.6b shows the automaton for receiving PresetOff messages.



(a) Receive PresetOn

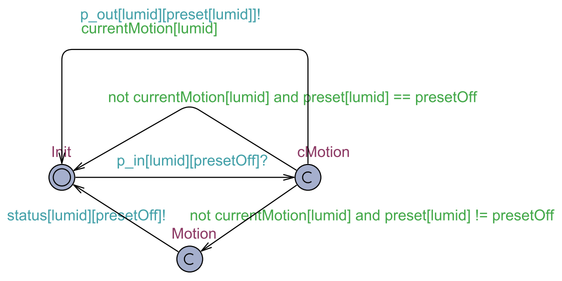

p_out[lumid][preset[lumid]]!
currentMotion[lumid]
not currentMotion[lumid] and preset[lumid] == presetOff
Init
p_in[lumid][presetOff]?
cMotion
status[lumid][presetOff]!
not currentMotion[lumid] and preset[lumid] != presetOff
Motion
(b) Receive PresetOff

Figure 5.6: Luminaire status automata

5.3.4 Preset messages

Preset messages should be sent when motion starts and when no motion is detected during the hold time. A point of interest is the automaton for sending the PresetOff message. Figure 5.7 shows the model in Uppaal. The luminaire can be in the NoMotion state for as long as the *tNM* (discrete) clock is lower than the hold time. Due to unpredictable latency in the test setup, the hold time in the model is extended with additional *noMotionSpanMax*. The value of this parameter is set to a small value such as 1 or 2 seconds, depending on the hold time. The minimal hold time is 30 seconds, adding a few additional seconds is acceptable behaviour for the IUT and it avoids failing the test due to small unexpected delays in the test setup.
Other notable behaviour is that when the luminaire changes to the PresetOff state, because it has received a PresetOff message from an other luminaire, the luminaire should no longer sent the PresetOff message.

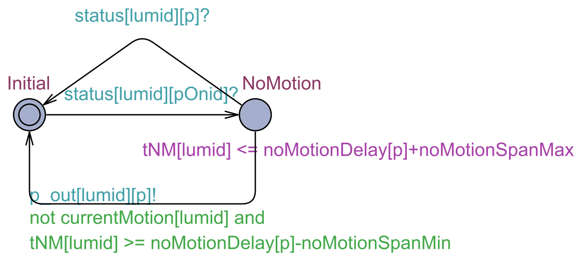

status[lumid][p]?
Initial
status[lumid][pOnid]?
NoMotion
tNM[lumid] <= noMotionDelay[p]+noMotionSpanMax
p_out[lumid][p]!
not currentMotion[lumid] and
tNM[lumid] >= noMotionDelay[p]-noMotionSpanMin

Figure 5.7: PresetOff automaton

5.3.5 Execution of experiments

One of the frustrations experienced during testing appeared with observing network traffic. For unknown reasons the test-adapter (PrismaUI) sometimes stopped observing broadcast messages from the luminaires. This was resolved by sending a broadcast message from the PrismaUI itself. This does require that a broadcast message is sent whenever the observing has stopped. To address this problem, we sent a *KeepAlive* message from the PrismaUI periodically during the testing. However the problem does not occur at specific times but rather random, which means that the test execution has to be observed manually, to make sure that PrismaUI remains observing communication between the luminaires. Note that if the PrismaUI does not observe the preset messages, the test will fail because, according to the tester, the luminaire changes to a different preset status without observing a preset message or occupancy. It is recommended to automate sending keepAlive messages periodically when continuing with model-based testing in the Prisma project.

5.3.6 Test results

In total, over 30 hours of testing has been performed on the normal behaviour of the system. During these tests, no unexpected differences between the model and the IUT have been detected at the end. Most of the test failures were caused by time delays caused in the test setup. To give an idea of the test cases, in 1 hour each luminaire produced about 30 motion events and the luminaires went to the PresetOff state around 10 times. It is worth mentioning that during the tests, the models have been adapted depending on test results and observations during the testing. The result of this, is a model that appeared to be corresponding with the IUT. We cannot guarantee that the model is correct due to the infinite test cases, but we assume that the time of testing is long enough to cover most cases. When confident enough about the model, we continued with robustness testing.

5.4 Robustness testing

After testing the normal behaviour of the system, the models and test setup have been extended to test the robustness of the system. The robustness aspect that is being tested is message loss. As we have seen in Section 4, robustness against message loss is considered in the category **Corrupt, lost, or delayed communication**.
In terms of changing the test setup, an additional network switch is included in the physical setup. This makes it possible to have a separate network for each of the two luminaires, to allow 'filtering' of communication between the luminaires. The software is extended to randomly forward and drop messages between the two networks.

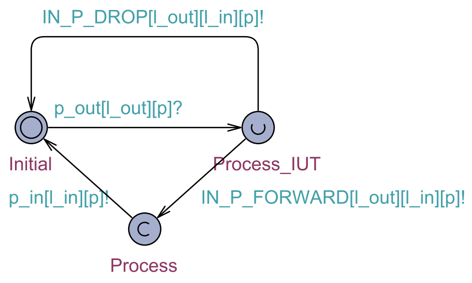


Figure 5.8: Network automaton, tester in control of environment

5.4.1 Modelling message loss

Section 3.5.7 provides two methods for using model-based testing to test against communication failures. In the experiments we have tried both approaches.
In the first approach, the tester decides which messages are lost. The tester observes messages from one luminaire and provides them as input for the other luminaire. Due to hidden information, we cannot reproduce a message in the test adapter. If we were to let the tester decide if a message is lost or not, we have to (temporarily) store them in the adapter. This can be done by means of a message queue. To keep the message queue synchronized with the tester, the tester should also inform the test adapter when a message is dropped.
Figure 5.8 shows an example of an environment automaton for the network of preset messages. The capital channels, **IN_P_FORWARD** and **IN_P_DROP** are inputs for the adapter to either forward or drop preset messages. The non-capital channels are internal channels in the automata network. The proposed approach requires that the tester can, without any delay, provide input to the IUT after receiving output.
In order to let the tester control message loss, additional delays are introduced in the process because, for each message, communication with the tester is required. In the case of Tron this communication goes via TCP. The tester should decide on message loss, as soon as, a message arrives at the tester. In practice this turned out to be difficult. To let the tester, in our case Tron, produce input with precise timing, the test tool must not be delayed by external dependencies, such as the OS scheduler. In our case, we ran into this scheduling issue which made it difficult to continue with this approach. For this reason, the second method was used. In the second method, the model and tester have no knowledge whether a message is lost or not. Instead the environment is controlled and communication is either randomly or in a controlled manner altered. For the model this means that the network automaton has to be non-deterministic to either allow observing message, or not observing messages.
The network automaton of preset messages is given in Figure 5.9. In this automaton, whenever a preset is produced by luminaire **l_in**, **Process** becomes the active state. In the **Process** state, the model can go back to the initial state, by either forwarding the message to luminaire **l_out**, or not forwarding the message. A problem found with this

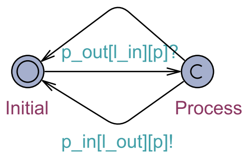


Figure 5.9: Network automaton, real environment in control

approach is that the state set of Tron grows heavily due to the combination of discrete clocks and non-determinism. For every message, there are two possible resulting states. Either the message was observed by the luminaire or it was not observed. In the case of synchronization messages, the clock value of one luminaire is copied to the clock of the other luminaire. This means that for every message, there is a new possible value of the clock. Either the clock stays as it is, or it receives a new value from another clock. The large state set resulted in heavy calculations and eventually Tron was unable to continue testing.
In a long period where luminaires do not return to the PresetOff state, Tron cannot rule out states with different clock values. Tron can only rule out states when a presetOff message is observed, only then the clock has reached the hold time which means that only states with a clock value equal to the hold time are possible states. All other states with different clock values can then be dropped.
To avoid this, we have used a workaround where the tester is not informed when the luminaire produces a synchronization message which was dropped in the network. This means the tester only observes synchronization messages that are not lost. The result is that there is only one possible new state after a synchronization message. This workaround made it possible to test behaviour after messages loss.

5.4.2 Inconsistent states

A point of interest in the analysed lighting system, is the time span in which the system is in an inconsistent state, as well as whether the system eventually recovers. To validate that the system does not stay in an inconsistent state, longer than allowed, an observer automaton, shown in Figure 5.10, is used. The observer automaton works similar to the observer automata in [13] except that the automaton also includes time invariants. The automaton goes to the inconsistent state, as soon as, the preset status of the luminaires is not consistent (all luminaires are in the same preset state). When the system changes to a consistent state, the automaton leaves the *Inconsistent* state. If during the test, the system is longer in the *Inconsistent* state than the allowed time invariant, the test will fail.

5.4.3 Execution of experiments

In the execution of the tests, the second method for message loss is used, where the tester has no knowledge whether the message was dropped or not. In this approach the

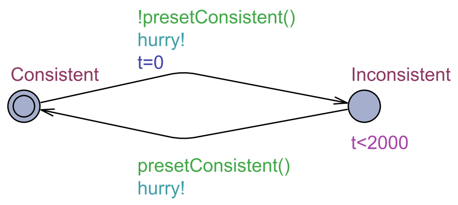


Figure 5.10: Inconsistent state automaton

model is highly non-deterministic. When executing the tests, we found that the state set of Tron was becoming too large to continue testing. The reason for this, is that due to the non-determinism in the model, combined with the discrete clock, the number of possible states was increasing heavily when time passes. This is not much of a problem in the case of preset messages, because they are sent only a few times. More problematic are the synchronization messages, as they are sent periodically. A workaround is to not allow synchronization message loss in the model and thus removing some of the non-determinism. Instead, whenever the IUT produces a synchronization message and the PrismaUI decides to drop it, the message is not considered output of the system and therefore not send to Tron. This means that, in the model, all the synchronization messages that arrive as output from one luminaire, are received as input for the other luminaires.
In total, over 20 hours of testing has been performed to test the behaviour after message loss. In these test, messages are dropped randomly, with a probability of message loss ranging from 1.0, where all messages are dropped, to 0.01, where only a few messages are dropped. In addition, tests are performed where only particular messages are lost. For example, in one test, only synchronization messages were dropped. In terms of motion start and end events, this was similar to the testing of normal behaviour. In one hour each sensor detected around 30 motions. Depending on the test, around 10 times the luminaires were both in the PresetOff state. In some tests however, only one luminaire was in the PresetOff state due to loss of messages, which is discussed in the next section.

5.4.4 Results

During the tests, no new, unknown problems in the system caused by message loss were found. In the tests, known cases where the system did not recover within the allowed time span were found. The most basic case is where the PresetOn message is dropped. In that case the luminaire, that did not receive the PresetOn message, is in a different preset state and does not go to the correct state because it only receives synchronization messages; synchronization messages do not include the preset state but only the clock value.
Another scenario that was found is that, when all the synchronization messages are dropped, the luminaire, that did not see motion, switches off while the other luminaire is still detecting motion. However, the probability of all the synchronization messages

being lost is close to zero.
It is not surprising that no new inconsistencies were detected with model-based testing. The model used in MBT is constructed from a model that is previously has been analysed with model checking in the Prisma project. The inconsistencies that were found were validated by replicating them on the real implementation. The advantage of model checking is that all possible states in the model are evaluated whereas in MBT only part of the total state set is reached.

5.5 Validation of Prisma models

To make the Prisma model usable for model-based testing with Tron, some additions to the model had to be made. First of, the model had to be adapted to make it usable for model-based testing. This means that the model has to be extended with an environment. The environment determines when input and output is possible from the IUT. Apart from the environment, the models have to be extended with the test setup. The Prisma models are models from the IUT and do not contain additional timing caused by the test setup. The test setup should at least contain adapter models that translate internal channels to external input/output channels. Furthermore the models have to be adapted in order to be supported by the older Uppaal version in Tron, as explained in Section 5.2.
A problem found during the experiments is that the control of the sensor does not occur real-time. There tends to be a small delay between sending the MotionStart event and observing that the sensor actually sees motion. This is a problem for deciding when the system is allowed to produce the PresetOff message. The PresetOff message should be sent after the hold time has passed since the last motion has been detected. If there is a delay between the tester generating motion and the IUT processing this motion, then the model will also expect the PresetOff message sooner. One way to solve this is to allow more time in the model to receive the PresetOff message. A different solution is to observe in the model when the occupancy sensor actually detects motion. Both methods are not optimal. The problem with extending the allowed time is that we cannot observe when the IUT has produced the PresetOff message too late. The problem with observing the occupancy sensor is that observing of the status change, as explained in Section 4.4.3, can be delayed by 400 milliseconds and therefore be observed after the PresetOn message was observed.
As mentioned before, the model used for model-based testing is straightforwardly derived from the Prisma model. No differences in the behaviour specified in the new model and the behaviour of the IUT were found. For this reason we conclude that the Prisma model is a valid representation of the lighting implementation. Therefore, it was assumed that the scenarios that lead to an inconsistent state, found with model checking, are present in the real implementation. The scenarios found with model checking have been confirmed by reproducing them in the implementation.

6 Discussion

This section discusses additional findings of the research. This includes an evaluation of the performed experiments and how well model-based testing can, with the current tools, be used in the industry. In addition the techniques of model checking and model-based testing are compared, when used to analyse lighting systems. Finally, the use of model learning is discussed for learning a model of the lighting system.

6.1 Model-based testing of indoor lighting systems

In the experiments, the lighting system was tested with model-based testing. There were no particular problems in connecting the MBT-tool to the existing test setup. Off course the software of the luminaires has to support the extra functionality, such as observing the light output level, that is required to observe the output of the system. This added functionality however, is something that is also needed with other test techniques.
As mentioned, the light level output and motion sensor status are not observed real-time. This is something that would be preferred for model-based testing. The software for testing however, is in full control and can be designed with keeping model-based testing in mind.
In general the reactions, of the people involved in the development of the lighting system, to a model-based testing approach were positive. The idea of automated test generation, combined with validation of the behaviour with model checking in an early phase, is something that will contribute to the development process as well as to the quality of the product. With model-checking more faults in the specification are found because all possible behaviour can be checked. This improves the quality of the product. One important remark however, is the lack of coverage statistics. Coverage is an important measurement to decide what part of the system is not tested and requires additional tests. In the current tools, particularly Tron, this is something that needs more attention. Coverage in tools is further discussed in Section 6.3.

6.2 Results of the experiments

The experiments that were performed did not result in the findings of new unknown inconsistent states. This is not surprising because the model that was used was constructed from a model that was used in model checking, where the behaviour was analysed when a message is dropped or delayed. Due to the real time in the system, particularly the hold time, the number of states of the model that are visited with model-based testing is much smaller than the number of states that are visited in model checking. This means

that new, unknown inconsistencies would only be found if the Prisma model is different from the behaviour in the implementation.
Not all the scenarios that were found with model checking appeared in the tests in model-based testing. The reason for this is that the MBT-tool, Tron, randomly decides when to generate input. This means that if the environment model, which defines when and what input and output is accepted, is non-deterministic, then it is possible that parts of the model are not visited during the test. To increase the model coverage, the test time can be increased. This should increase the probability of each transition because the states are visited more often.
Because most of the scenarios (found with model checking) that lead to inconsistencies require very specific behaviour, the probability of it occurring during the test, is rather low. To test scenarios, found with model checking, the environment model can be made explicitly to only allow the behaviour of the scenario. Because the found scenarios were validated manually, the validation was not repeated with model-based testing.
A more important result, rather than the finding of no inconsistencies, is that no differences between the model and the behaviour of the implementation were found. The used model is constructed from the Prisma model where the general behaviour is modelled in the same way. We therefore conclude that the Prisma model is a valid representation of the implementation and thus the model can be used to further analyse the lighting system.

6.3 Tools

In this research the tool Uppaal Tron has been used. One of the things that is missing in Tron is a graphical representation during the execution of the test. In our research, the model was constructed from the implementation, which meant that experiments had to be done to ensure the correctness of the model. It would have been nice to have a graphical representation of the current state set of Tron. This is in particular an advantage when a test fails. In the current version of Tron, the final state is given as output in the form of text. This means that in order to understand the reason why the test failed, the text has to be converted, manually, to a state in Uppaal. An argument to why there is no graphical interface, given by the creators of Tron, is that a graphical interface results in unnecessary delays on the test machine. I agree that this is a valid argument but during experiments, a graphical interface will highly improve productivity as the test output does not have to be converted manually when a test fails. To address this problem, I have created a small simple graphical interface in Python that converts text output from Tron and displays the current state in the Uppaal model. This gave us an approximation of the current state in the model and made it faster to understand why a test failed. Figure 6.1 shows the graphical interface with the model of the lighting system loaded. The current state is highlighted in red and the values of the local variables are listed in the right top corner of each automata.
Furthermore, as mentioned before, the development of Tron is not in line with the development of Uppaal. This requires that the model has to altered in order to be

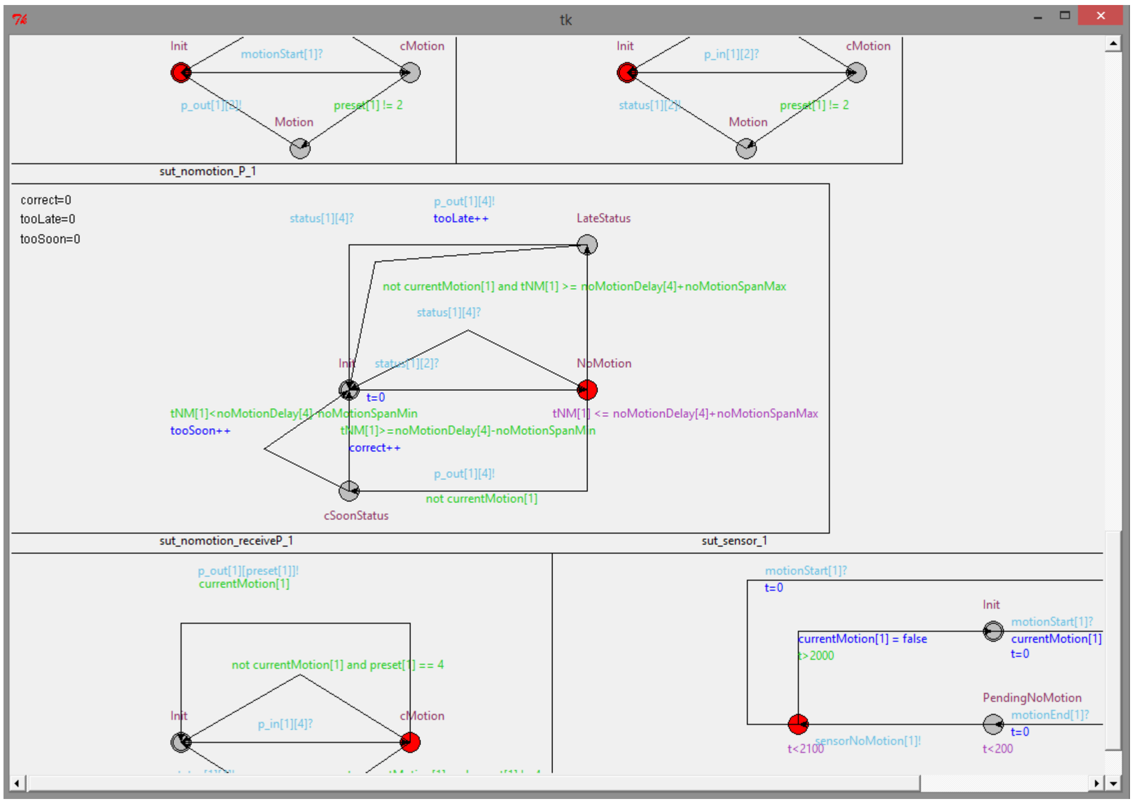


Figure 6.1: Graphical interface for Tron in Python

used with Tron. In particular, copying clock values is something that is critical for the Prisma project. With an altered, second model, it is possible that the two models are not precisely the same. For example, adding an edge to remove guards on receiving channels, as shown in Figure 5.1, means that there is an extra transition needed to achieve the same result. The extra transition introduces new states that are not considered with model checking. This particular example can be avoided by not using guards on receiving channels in the model, used for model checking. Nevertheless, these kind of incompatibilities lead to unnecessary frustrations. When combining model checking with model-based testing, the models of the system should not require addition changes. Off course this does not include additional models, such as environment models in Tron, that are required for model-based testing. These added automata do not affect the model of the system.
One of the things that is currently missing in Tron, but more general in model-based testing, is test coverage. In non-timed tools, such as jTorX, coverage is provided by means of transition or state coverage. When considering time, revisiting states can result in different behaviour of the system because the timing is different. This makes it difficult to say something about coverage. In Tron, coverage is given by means of

time, coverage increases by extending the time of testing. Tron should provide some form of coverage in the model. This could for example be statistics, how often a state is visited and how often a particular transition was taken. Coverage of the system is an important measurement for the industry to verify the correctness of the system after the execution of the test. It also provides information about which part of the system requires additional testing.

6.3.1 Yggdrasil

During the experiments I have explored a new feature in Uppaal (4.1.19), Yggdrasil. Yggdrasil is an offline test generation tool that generates test cases from test code, entered in the model [32]. Within the Prisma project, Yggdrasil can be an interesting tool, as it can generate test code from traces that are found with model checking, however, time is not considered in the test generation. Yggdrasil is still in an early stage of development, one of the issues is that automata cannot use local variables in the test code generation. This is a critical feature when the model contains multiple instances of the same automaton, which is the case in the Prisma model. This feature might be added in a future version of Yggdrasil [23].

6.4 Model checking and Model-based testing

Model checking and model-based testing are two techniques with different goals. The goal of model checking is to validate whether particular behaviour always holds, whereas the goal of model-based testing compares the model with the real implementation. Preferably these techniques are used in combination. However, the techniques are also applicable separately. This section describes advantage and disadvantages of both techniques when applied in the Prisma project.

One of the notable differences in the execution of model checking compared to model-based testing, within the Prisma project, is the shorter time of execution. Online model-based testing requires the system to function in its normal behaviour. The lighting system that has been analysed, typically has a hold time of 10 minutes or more. This means that testing takes a large amount of time. With model checking, time is simulated and thus, trying different paths in the model is much faster.
In model-based testing the model is checked against the real system. If the system produces different output than the model, then either the model is wrong or the system is wrong.
With model checking the model is not compared to the actual system. This means that replicating a trace of inputs on the real system may result in different behaviour. This also means that model checking may not find faults in the implementation because the model does not contain these faults.
One of the difficulties that was found with testing the lighting system is that the real system, within the test setup, may contain delays that are not considered in the de-

sign. In the lighting systems these delays occurred in the communication between the luminaires, as well as between the test adapter and the physical sensor simulator. This makes it difficult to take the models that were used in model checking, and use them for model-based testing.
It is important that the system is designed in such a way that a tester adapter can be connected to the real IUT without causing too many delays, or at least the delays should be possible to model.

6.5 Model learning

Within the Prisma project, models were created manually from an informal specification. A different technique to do so is model learning. With model learning, a model is constructed from a number of tests. The advantage of model learning is that the model is always conform the actual system. To learn a model from the system, a sequence of inputs is executed on the system and the output after each input is observed. The learned model can then be used to apply model checking to identify inconsistencies between the model and the specification. In comparison to the approach of model checking combined with model-based testing, described in Section 3.4, model learning replaces model-based testing. Instead of validating the correctness of the implementation, the model is learned from the implementation.
The tool Learnlib has been used to experiment with learning the lighting system [25]. A critical aspect of this particular system is the real-time behaviour. Timing in model learning introduces a large non-determinism in the behaviour of the system. It also introduces outputs that occur in the future instead of right after an input is supplied. For instance, in the lighting system, the lights go off after a certain hold time. For model learning this is a problem because the learner has to relate the current output to one of the inputs that was produced somewhere in the past. This is particularly a problem if the system has noise in the real-time behaviour.

In the lighting system, we also have the problem that the order of outputs can be different, as explained in Section 4.4.3. This non-determinism makes it impossible to apply learning with the currently available algorithms.

7 Conclusion and Future Work

This Section concludes the research. In Section 7.1 the research question is reviewed. Section 7.2 provides suggestions for future work.

7.1 Research question

To answer the main research question, I first answer the sub questions.

What is considered robustness in system engineering?

A number of different definitions from the literature that define robustness are discussed in Section 2.1. There is no precise definition that all literature agrees upon. A summary of the definition of robustness of systems can be defined as:
A robust system should remain operating correctly after unexpected input or after changes in the environment or internal structure.
What is considered correct behaviour, has to be defined in the design. A system can only be robust to a certain degree. In order to define to what extend a system is robust, robustness has to be considered in the design.

How can robustness be categorized in terms of testing?

In Section 2.4 a number of categories and characteristics are given that characterize robustness of software systems. Robustness is divided in two main categories, namely, input, and environment and internal structure. Within the input category there are three types of content that can be distinguished and require a different approach in testing. The first one is valid input that the system accepts but is provided when the system does not except it. The second one is input that is not within the domain of the input of the system. To define what is input of a system, a specification is required. The specification defines what is input that is accepted by the system. Finally there is malicious input. Malicious input is input outside the domain of input of the system but requires a different approach in testing. Malicious input often contains input that is very close to the accepted input whereas other, out of domain, input can be constructed randomly.
Changes in the environment and internal structure are considered to result in the same effects on the behaviour of the system. Robustness is categorized by the consequences rather than the cause of failures or changes. Consequences that are distinguished are effects on, system elements, behaviour, data, communication and time.

How can model checking be used to analyse robustness?

Robustness covers a large part of system behaviour. It is not possible to propose one general method for all robustness problems. In order to analyse robustness, it has to be decided what aspect of robustness is analysed. For this, the proposed categorization can be used. For each of these different categories, methods are proposed in Section 3.5 whether model checking and model-based testing can be used. Not all categories can be analysed with model checking. This is particularly the case in categories that are related to invalid input that is not within the domain of the input of the system. In model checking the model is closed in terms of input and output. The model does simply not include invalid input. In the category where input outside the domain is considered, input has to generated in order to analyse the system with invalid input. The generation of the input has no effect in model checking because the remainder of the model does simply not include the generated input.
A real system however is input enabled and in theory, anything can be given as input. In model-based testing the model does not have to be closed in terms of input.
In the case of robustness against environmental and internal changes, the model used in model checking can be altered or extended to analyse the behaviour when an element of the system or environment does not operate as expected. By means of properties, the behaviour in the model is analysed.

How can model-based testing be used to test robustness?

As with the previous question, Section 3.5 provides a method to use a model-based approach for testing robustness. In contrast with model checking, model-based testing is applicable in all categories. A general advantage of model-based testing over normal testing is that, in model-based testing, it is much easier to introduce faults or invalid input during normal execution. In model-based testing, the model defines how test cases are constructed. Invalid input can be inserted half-way through the trace of valid input. The MBT-tool can observe the system to validate that the system stays within the allowed behaviour, specified in the model.
To test changes in environment or internal structure, there is one general approach. The environment in wich the system is tested can be altered to randomly or in a controlled way, insert failures or changes in the environment. The IUT should remain conform to the model even though the environment does not cooperate. Some of the proposed categories have additional methods that can be used. These are explained in Section 3.5.

What are the strengths and weaknesses of model-based testing compared to model checking for validating robustness of a system?

In Section 3.5 advantages of model-based testing and model checking are given for each category in robustness. In some categories, model checking is the preferred approach. In model checking a larger part of the behaviour of the system is checked whereas in model-based testing, only a subset of all possible states are checked. The advantage of model-based testing is that the changing environment is not simulated but the real environment is considered. Testing robustness in model checking requires that failures in the environment are modelled. The pitfall of this is that failures are formed from human intuition. In model-based testing, the real environment is tested and thus includes all failures that can occur in the system. Nevertheless it is possible that these changes in the environment do not occur in the test execution. Furthermore another advantage of model-based testing compared to model checking is that model-based testing performs better in terms of scalability. In case of lighting system, adding more luminaires to the system massively increases the number of possible states. In model checking this will lead to a state-explosion. In our experiments, model-based testing also resulted in a state explosion. However the reason for this was the construction of the model, namely the discrete clocks. Without the discrete clock, the state explosion would not occur.

Is the Prisma model a good representation of the real implementation?

I have performed a number of tests to validate the model under normal behaviour of the system. Due to limitations and difficulties in applying model-based testing with Tron, a new model is used that is formed from the Prisma model. In this model no differences in behaviour compared to the real system were found. We therefore assume that the basic behaviour in the Prisma model is correct with respect to the implementation.

Which model-based testing tools can be used to test indoor lighting systems?

An important part that is analysed in the lighting system is the hold time. Luminaires should switch off after no motion has been detected for at least the hold time. This means that the model includes time. The MBT-tool should ideally include time. One of the MBT-tools that supports time is Uppaal Tron. In Tron, Uppaal models are used to describe the behaviour of the IUT. A particular advantage of Tron is that the existing Prisma models, that were used in model checking with Uppaal, require no additional steps to convert between two modelling languages. During the experiments, we found that the Prisma models were too precise in timing and it would be easier to construct a new model. Another problem found during the testing is that, in order to model the system, clocks in the model have to be copied, which is not supported in Tron. A workaround that was used, is to include a discrete clock in the model. This discrete clock could have also been modelled in another non-timed tool, such as jTorX, by letting an adapter periodically produce a tick output as part of the output of the IUT. An advantage of jTorX is that the verdict of the test includes transition coverage. Additional research can be done to explore whether abstracting from time and use a discrete clock is possible for the lighting system.

To what extend has the Prisma model, used with model checking, be changed to make it usable for model-based testing?

To make the Prisma model usable for model-based testing, the model has to be extended to include the input and output of the system in an environment model. The environment model defines when the tester (Tron) is allowed to provide input to the IUT and output from the IUT is observed. One of the issues found in the experiments is that the Prisma model is very precise in timing. For instance if the hold time is set to 20 seconds, the model considers the luminaires to go off precisely after 20 seconds. In the real system there are often some small delays. The end of the hold time should not be a precise timing but rather an allowed time span.
Apart from functional changes, the model also has to be converted to make it usable with the MBT-tool. In the case of Tron, this is limited to the removal of new features in Uppaal that are not supported in Tron. For other tools, the Uppaal models might have to be converted to a different modelling language.

How can robustness against message loss be tested with model-based testing?

Message loss is a robustness aspect that is the result of an unreliable environment. More precisely, message loss is considered in the robustness category, loss of communication. In Section 3.5 two methods are proposed to test against loss or corrupt communication. One method considers messages as input of the system. Because the tester is in control of the input that is provided to the IUT, the Tester can decide when messages are dropped or delayed. The other method considers the environment to be controlled, in the form of randomly dropping or delaying messages. The model contains only the allowed behaviour of the system. With model-based testing, the IUT is validated on conformance to the model when not all messages are received. The model is non-deterministic in the sense that it either considers messages to be send normally or messages being dropped. This latter approach is used in the experiments to test the robustness against message loss of indoor lighting systems. The results show that model-based testing can be used but has the limitation that only a limited number of situations are tested, whereas in model checking, a complete set of possible states are checked.

How can robustness against power loss be tested with model-based testing?

Power loss is considered to be a failure in the environment that leads to loss of input. Power is an input of the system. In Section 3.5, a method is proposed to test against missing input. To test loss of power, first input of power has to be converted to events. This can be done by considering an event when the system starts receiving power and an event when the system stops receiving power. Power loss can than be analysed by modelling an environment that simulates the power supply in an unreliable environment. In the experiments, power loss has not been analysed. Simulating loss of power, to perform model-based testing, requires additional hardware to intercept power to the lighting system. Model-based testing of power loss is considered future work.

After answering the sub questions, the main research question can be answered.

What are the different types of robustness in system engineering and can they be tested using a model-based approach, furthermore, is model-based testing a useful method to test robustness of indoor lighting systems?

In the thesis, a number of robustness categories are proposed for system engineering. The categories range from invalid input to changes in environment or internal structure. In comparison to manual testing, proof can be provided, with the use of model checking, that the system stays within the correct behaviour. With model-based testing the implementation can be validated with the model. For each category a method for a model-based approach, where model checking and model-based testing are considered, is proposed. The experiments that are performed, show that model-based testing itself does not add much extra value in the finding of inconsistent states in the lighting system. The reason is that, with model-based testing, only a limited number of states in the state set are reached, whereas in model checking, all possible states are reached. The added value of applying model-based testing, is to gain confidence in the correctness of the implementation with respect to the model.

7.2 Future work

This section provides suggestions for future work. Research is needed to further extend and evaluate the proposed methods for model-based robustness analyses. Within the Prisma project, additional work can be performed to further analyse the system.

In robustness analysis, more experiments with the different proposed methods should be performed in the industry to validate the usefulness. The experiments in this research were limited to model-based testing of loss of communication. Other proposed methods are very different from the used method in the experiments. For instance, testing invalid input introduces transitions where input is generated randomly. Experiments should show whether this method can be applied and how useful it is in practice.
In the proposed methods of testing malicious input and input outside the domain, the generation of input is not considered. The proposed methods can be extended to include automated generation of invalid input. It is possible that existing techniques, proposed in the literature, can be used. The generation of invalid input was not in the scope of this thesis. The same holds for methods in the categories where faults are inserted in the system. Additional work is needed to extend the proposed method to automate the process of inserting faults in the system.
More attention should be given to separate the system and the environment. I have provided one approach that is based on my own experience and intuition. It is possible that my approach is not applicable for all systems and still leaves room for misinterpretation. A full in-depth study should be performed to, ideally, define a formal definition of a system and its environment.

7.2.1 Prisma project

The work in this thesis is limited to testing basic behaviour of the lighting system. The Prisma model contains more functionality. Extending the model used for model-based testing, is something that can be done to further analyse the correctness of the Prisma model. In addition, power failures can be included in the analysis. For model-based testing, this does however require physical hardware to control the power supply via the test software.
The lighting system considered in the Prisma project is a highly configurable system. There are many system configurations possible, not only in the number of luminaires and sensors, but also in the configuration of the presets, building topology, deployment, and network topology. An automated process is preferred to create models of different configurations. This can, for instance, be achieved with a Domain Specific Language (DSL). The models can than, preferably in an automated way, be used for model checking and model-based testing. This makes it possible to easily analyse different configurations of the system.

In this thesis, the tool Tron has been used. The reasons are that time is a critical aspects of the Lighting system and the existing Prisma model is an Uppaal model. During the experiments we found that clocks had to be made discrete, in order to copy hold time timers from one luminaire to another luminaire. A discrete clock can, in theory, also be constructed in a non-timed MBT-tool by letting the adapter (PrismaUI) increase the clock periodically. The experiments could then be repeated with a different MBT-tool. Although I do not expect other tools to find inconsistencies, other tools provide 'better' results. The tool jTorx for instance, provides transition coverage, which gives some information of the size of the total functionality of the system that has been tested. Furthermore, jTorX does contain a graphical representation during the execution of the tests. The lack of a graphical representation is something that felt missing in Tron, as explained in section 6.3.
One of the topics that can be of interest in the Prisma project is using Tron to observe behaviour of the real system. It would be interesting to see whether the system stays in the allowed behaviour in the model, when the system is observed during real use. Tron could be used to monitor input and outputs and validate them in the model. This would allow to test real scenarios instead of scenarios that are defined in a test. This however requires a version of Tron where the test time is infinite.

8 References

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[2] Craig R Allen. Ecosystems and immune systems: hierarchical response provides resilience against invasions. *USGS Staff–Published Research*, page 7, 2001.

[3] Paul E Ammann, Paul E Black, and William Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.

[4] Alessandro Armando, Giancarlo Pellegrino, Roberto Carbone, Alessio Merlo, and Davide Balzarotti. From model-checking to automated testing of security protocols: Bridging the gap. In *Tests and Proofs*, pages 3–18. Springer, 2012.

[5] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[6] Jack W Baker, Matthias Schubert, and Michael H Faber. On the assessment of robustness. *Structural Safety*, 30(3):253–267, 2008.

[7] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[8] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270. Springer, 2010.

[9] Jean M Carlson and John Doyle. Complexity and robustness. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2538–2545, 2002.

[10] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking.* MIT press, 1999.

[11] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd*

IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, pages 31–36. ACM, 2007.

[12] Richard Doornbos, Jacques Verriet, and Mark Verberkt. Robustness analysis for indoor lighting systems. In *10th International Conference on Systems*, Barcelona, Spain, April 2015.

[13] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A model-based approach for robustness testing. In *Testing of Communicating Systems*, pages 333–348. Springer, 2005.

[14] Anders Hessel, Kim G Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal methods and testing*, pages 77–117. Springer, 2008.

[15] Santa Fe Institute. Robustness in natural, engineering, and social systems, http://discuss.santafe.edu/robustness, 2000-2003.

[16] ISO/IEC. ISO/IEC 25010 system and software quality models. Technical report, 2011.

[17] Erica Jen. Stable or robust? what's the difference? *Complexity*, 8(3):12–18, 2003.

[18] S.J. Kapurch. *NASA Systems Engineering Handbook*. DIANE Publishing Company, 2010.

[19] Kim G Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306. ACM, 2005.

[20] Paulo Leitão. A holonic disturbance management architecture for flexible manufacturing systems. *International Journal of Production Research*, 49(5):1269–1284, 2011.

[21] B. Meyer. *Object-oriented Software Construction*. Object-oriented programming. Prentice Hall PTR, 1997.

[22] Marius Mikucionis. Personal communication, March 2015.

[23] Petur Olsen. Personal communication, May 2015.

[24] I. Parmee. *Adaptive Computing in Design and Manufacture VI*. Springer London, 2011.

[25] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *International journal on software tools for technology transfer*, 11(5):393–407, 2009.

[26] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *Formal Modeling and Analysis of Timed Systems*, pages 250–264. Springer, 2008.

[27] Ali Shahrokni. *Software Robustness: From Requirements to Verification*. Chalmers University of Technology, 2013.

[28] Adenilso Simao and Alexandre Petrenko. Generating complete and finite test suite for ioco: Is it possible? *arXiv preprint arXiv:1403.7261*, 2014.

[29] Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[30] Sunita Tiwari and Arpan Gupta. An approach to generate safety validation test cases from uml activity diagram. In *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, volume 1, pages 189–198. IEEE, 2013.

[31] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

[32] Uppaal. Uppaal 4.1.19 manual - yggdrasil. Accessed: March 2015.

[33] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[34] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. *Model-based testing for embedded systems*. CRC press, 2011.