MASTER'S THESIS | LUND UNIVERSITY 2014

Automatic timing test of physical access control systems

Jonas Klauber

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2014-40

Automatic timing test of physical access control systems

Jonas Klauber
<jonas.klauber@gmail.com>

December 15, 2014

Master's thesis work carried out at Axis Communication AB
for the Department of Computer Science, Lund University.

Supervisors: Per-Daniel Olsson, <perdaniel.olsson@axis.com>
Emil Selinder, <emil.selinder@axis.com>

Examiner: Flavius Gruian, <Flavius.Gruian@cs.lth.se>

Abstract

The testing of a real-time system involves more challenges than in a regular one. Real-time system testing must deal with issues such as concurrency and timings. The difficulty increases further if the tests should be automated. Automatic tests tend to be static and use identical configurations for each test run and therefore have problems dealing with mutable parameters such as time.

This thesis presents a method for generating automatic testing of an event-triggered real-time system by using a model-based testing approach. Model-based testing is focused on comparing a system under test to a model. The comparison consists of automatically generating tests from the model, executing them, and comparing the output to the expected result. In the model variables such as time and probability are included in order to mimic real-life usage and deal with the problems in classic automatic real-time system testing.

Keywords: Automatic test, Model-based testing, Real-time systems, Event-flow Graphs, Markov chains

In memory of my grandfather Ulf, who always supported my studies but never got the chance to see the result. You will always be remembered.

Acknowledgements

First of all, I would like to thank the New Business department at Axis Communications for all your support, help and guiding during this thesis. An extra big thank you to my supervisors, Per-Daniel Olsson and Emil Selinder for their never ending patience and feedback. An extra thank to the QA department for letting me use their their Arduino software. I would also like to thank my supervisor and examiner at LTH, Flavius Gruian, who has been of great importance with his guidance and perspective. Also an extra thank you to Erik Westrup for reading and correcting the thesis. I want to thank my lovely girlfriend Midori Ng for the help with the language, support and putting up with us being separate during this project. Finally, I want to thank my family and loved ones for all your support during this thesis.

Contents

Chapter 1
Introduction

Today the technological development has solved many problems that in the past seemed impossible. We are now living in the midst of a technological revolution with a rapid increase of digital products and software in the daily life. New products and systems constantly get released to the market for us to buy. Some are made just for entertainment purposes, while others are essential for survival. In common for any kind of system, is that they have increased in complexity and have became more sophisticated over the years. Systems with a higher complexity makes the ability to create and carry out tests a lot more challenging. Today, when people as well as large companies rely on such systems, an error would be disastrous, and in the worst case lead to loss of life. No system is absolutely flawless and there is always a balance between the degree of testing and cost.

One important task for our system developers is to create and use tools for revealing errors and verifying the systems. This task is called testing and is ongoing during the whole development process and long thereafter. Testing is considered as an important task and often even critical. Developers today spend 30-50% of their time on debugging and repairing their systems [15].

According to studies, developers have produced a variety of testing techniques and methods to make things smoother, i.e. test driven development, model-based testing and automatic testing. Every method has its advantages and drawbacks depending on the system. An important attribute most methods try to make use of is **Automation**. Test automation is when standalone programs are developed for testing systems. They are capable of executing tests, reporting outcomes and compare results. Creating such a program requires a high effort during development phase, but once it is done, testing can be carried out repeatedly almost without effort at any time.

This thesis proposes a method to generate automatic test for an event-triggered real-time system. For evaluation we have carried out experiments on a physical access control system (PACS). The thesis will investigate the system and propose a method for generating a model to carry out automatic tests for the system and discuss its pros and cons.

1.1 Problem Formulation

In the regular case, automatic tests tend to be very static. For each change of an input, a new test case would need to be created. Therefore, automatic testing has problems in dealing with mutable parameters such as time. For a real-time system, which is time dependent, testing such as a parameter is crucial, a real-time system usually has a numerous amount of inputs and timings. Testing those would require creation of many test cases, one for every change in input or timings, although many test cases would look very similar. The creation and maintaining all these test cases would be difficult and a resource demanding task. It would be almost impossible to keep track of what and which parameters have been tested and therefore to perform any kind of exhaustive testing.

Classic automatic tests usually run with identical configurations each test run and that might fail to trigger exceptions which would occur with test parameters outside of the normal scope. In order to mimic a real-life usage during testing, it is desirable to automatically test certain aspects. Aspects that are in the interest of this thesis are:

Tests with varying timings Real-time systems depend on the actual magnitude of the timings and various delays. In regular static automatic testing each variation of timings and delays would require its test case of its own. Instead it would be interesting to create automatic tests that dynamically can vary timings and delays for each test run.

Pre-recorded user scenarios for testing From a pre-recorded use pattern, e.g. a log-file, our method should be able to create tests that copy the user behaviour with its characteristic timings properties for execution. This enables recreation of scenarios that are found interesting, such as critical parts, unexpected system behaviours, bugs etc.

By having a few automatic and dynamic tests, a lot of resources can be saved when it comes to maintenance and development costs. By dynamic we mean tests that can change their input and timing for each test run. This would allow a better control of what has been tested and what needs more testing, so more resources can be put there to do more exhaustive testing and in the long term reveal more errors.

1.2 Proposed Solution

The solution we propose is a model-based approach, which creates a model of the usage of the system which is then used to generate test automatically. Each test has a variation of inputs and timings and it gets executed on the system. The test result then get compared to the model. By adopting a model-based test approach, a more effective testing would be done, since we would have no static tests to keep update instead only one model.

The system under test (SUT) is a event-triggered real-time system. To model a is a directed graph based on the events in the SUT and the timings between them. Each path from the graph will represent an individual test case in terms of inputs. Different paths has different inputs and the timings are generated dynamically for each test run.

The SUT interacts with an operator (the user) who supplies inputs with varying timings in between. Every time an input is supplied, one or many events are generated and stored in an event-log at the SUT. From the model we need to be able simulate one or many users in terms of usage and timings. Since the model is based on the event-log from the SUT, if the model is based only on events from one user scenario, it will generate a test identical to that scenario. The model then can be used as a test oracle. It can predicate an outcome, that is used for compare and evaluation of the test, to decide if it passed of failed. The test generation and test oracle decision should be as automated as possible for an effective and increased coverage of the testing, something the model-based allow.

1.3 Thesis Overview

The following is a brief account of the contents of this thesis.

Chapter 2 - Background gives a theoretical background to system testing and real-time systems. There is also an explanation of our system under test for the experiment.

Chapter 3 - Related work describes work that has been done within the same area such as different types of tools and techniques.

Chapter 4 - Experiment looks into the components, how they are connected, how they communicate and how the method can be used. A theoretical explanation of the model and its attributes also have a step by step explanation with figures of model creation, test generation and ending with the test execution and result evaluation on the real system.

Chapter 5 - Analysis Earlier chapters serve as a foundation for chapter 5. We will analyze and discuss the hardware, model, automation, delays and tests generation from the experiment in aspects like pros and cons, alternative solutions and improvements.

Chapter 6 - Conclusion summarizes this thesis. We end by giving suggestions of future work that can be made for improving this area.

1.4 Contribution Statement

In this thesis, we propose a method for generating automatic tests for event-triggered real-time systems. We use a model-based testing approach which consists of creating a model of the system environment, generate automatic tests and execute them. The model is based on a graph built of events in the system and the timing between them. We then traverse different paths from the model that is used to generate multiple test cases which then get executed and evaluated. This research makes following contributions:

- We present a way to model an event-triggered real-time system by include and represent timings in the model.
- We present a way to use the model for generating automatic dynamic test cases for event-triggered real-time system.

- We show how to use the models as a test oracle and evaluate the test results.
- We show how to apply real usage scenarios to the model in order to recreate them in detail.

Chapter 2
Background

In this chapter, we introduce the concepts and definitions that are used throughout this thesis. Section 2.1 will give an introduction to system testing and a few testing methodologies within our interest. Section 2.2 will be an introduction and brief summary of Markov chains, which are used in the model for test generation.

Our system under test (SUT) is a real-time system (RTS), therefore Section 2.3 is a generic presentation of RTS and work that has been done in terms of testing such a system.

Finally, Section 2.4 presents an overview of the SUT, the physical access system that was used during the experiments.

2.1 System Testing

Testing is considered to be one of the most important stages for a system with high reliability. System testing has two distinct goals [43]:

- To show to the customer and developer that a system meets its requirements.
- To identify incorrect behaviour in the system with respect to a specification.

Two important aspects of testing are validation and verification and according to IEEE are described as [45]:

Validation The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. "Are we building the right product?".

Verification The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. "Are we building the product right?".

The type of testing and the level of intensity differs for each project, being a balance between the risk of having an error and the cost of that error to occur. For example if your turtleRace program crashes it can be handled, but if a nuclear plant system has an error the outcome can be deadly.

Testing is a costly but essential task. Studies show that system testing takes up to 30-50% of the development and maintenance cost [17]. Developers have produced an endless number of testing methods and techniques to make the work more efficient. Today, one can see a combination of different methods depending on the type of project. Two important aspects for a test method are:

- Effectiveness - They should have a high probability of revealing existing faults.
- Effort - They should require a small effort to execute.

2.1.1 Testing Methodologies

Developers have came up with countless techniques and methodologies for testing. Here we will go through a handful of them which are considered to be important for this thesis. In general they are divided into two different groups: white-box testing and black-box testing. Within a project usually both methods are used in one way or another, sometimes referred as grey-box testing.

Black-box Testing

Can also go under the name functional testing. In this testing method the focus is on what outputs are generated from inputs. The system's internal states and mechanisms are completely ignored [45]. See figure 2.1.

Advantages:

- Test cases can be designed from the users perspective.
- Can be made by anyone, testers do not need to know anything about implementation.
- Test cases can be designed as soon as the specification is complete.

Drawbacks:

- Only a limited number of inputs may be tested, paths within the system may be untested.
- Without a clear specification it may be hard to create test cases.

An example of black-box testing would be to see if a car engine starts up with the correct key.

White-box Testing

White-box testing is also known as structural testing and glass box testing, the opposite of black-box testing. In this method the tester is required to have knowledge of system and its implementation and takes that into account during testing, e.g. testing a specific method in a software. With the possibility to observe the internal state of the system it is easier to work towards a certain goal or output [45]. See figure 2.2.

Advantages:

- Testing can start early, testing small parts of the system on the go.
- Testing is able to track and make sure more paths are covered.

Drawbacks:

- Requires the tester to have a detailed knowledge about the system.
- Hard to maintain if the implementation changes too frequently.

An example of white-box testing would be to test if a certain method returns the correct data type.

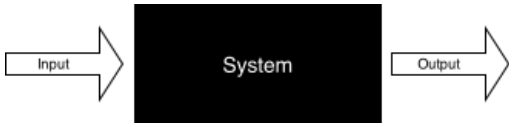

Figure 2.1: Black-box testing

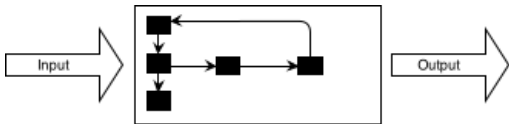

Figure 2.2: White-box testing

Test Coverage

As mentioned earlier there is a delicate balance between the cost for a error and the quality of the system. A question that arises in every project: "How much testing is needed to consider the system to be completed?"

Pressman states that one can answer those questions by collecting different metrics [36, p. 639-640]. The most common one is test coverage, which measures the amount of test cases covered in the system. The scale goes from no testing (none or only a few test case) to exhaustive testing (every possible test case).

Two other types of test coverage metrics are: code coverage and coverage of requirements. Beizer introduces different strategies to measure the coverages and describes their strengths and weaknesses. In general, the more test cases that are used, the stronger the strategy is [13].

Unit testing

Unit testing is a white-box testing method and improves the code coverage. It is made for testing the smallest testable piece of a system, a "unit" or "module" [13, p. 25]. In software it may be e.g. a function or a method. In this strategy, every unit is tested separately, but it does not test interaction between different parts of the system. Often the output from one unit is the input to another. This type of testing important since it makes sure each unit works separately, before they are connected together.

2.1.2 Functional Testing

Functional testing aims to test if the system functions work as specified. It is the process of quality assure (QA) a system based on its requirements. This can be a very demanding task since the whole design of the system has to be tested at once. Since the tests need a fully functional system, it usually takes place late in the development process [8]. There are many different types of functional testing such as Usability testing, Regression testing etc.

2.1.3 Test Oracle

A test oracle is the mechanism for determining whether a test has passed or failed. It is expected to provide the correct result(s) for any inputs, and do a comparison of the actual results and the expected ones. An oracle can be represented as the expected results, the process of generating expected results or a program, or a mechanism used to generate expected results, see figure 2.3. The oracle is not a test, but a test can use the oracle as source of correct and expected results. The ability to automate testing is constrained by the ability to create and use oracles for a system [16]. Hoffman [25] presents six main characteristics for using oracles in test automation which are shown in table 2.1.

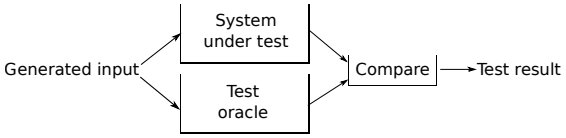

Figure 2.3: Testing generated input with a oracle.

In general, a more complete and accurate oracle will be more complex (less maintainable and more resource demanding). Therefore a common problem for a complex oracle is to keep it updated with changes in the SUT because it is harder to maintain. In some cases the oracle shares some parts with the SUT. System defects can then be missed due to the oracle generating and accepting a buggy part.

2.1.4 Automatic Testing

Traditionally, test cases were executed and evaluated manually. To manually maintaining and execute several test cases multiple times is time-consuming. It requires a lot effort in terms of man hours and therefore is inefficient. In order to solve this problem, developers

Table 2.1: Main characteristics of a test oracle [25].

Characteristic	Description
Completeness	Which input is covered? Which failures can be detected?
Accuracy	How similar is to the system under test? How independent?
Usability	How is the information given? Does it fit the intended use?
Maintainability	How complex is the oracle? How costly is it to update it?
Performance	How fast does it generate output? How often must be run?
Cost	How much does it cost to make it? How much to execute it?

have created tools that can execute tests faster and easier; or in other words, automatically. In most cases, the creation of these tools requires a lot of resources, but once completed, tests can be executed at any time with lesser effort and can easily be improved during the development stages. Overall, by automating tests, large amounts of resources can be saved and better used elsewhere [44].

Test automation consists of three major stepping stones: generating a test input, generating an expected output, followed by executing tests, verifying and reporting the results.

Input generation Generating an input is a very challenging task, and it is constructed based on the system's code or/and specification. There are different ways to do this, but for the purpose of this thesis, model-based testing is utilized as explained in section 2.1.5.

Output generation Output generation is critical as it provides a measurement for comparison with the system's output from a test. Some methods for generating outputs include pre-computed input/output pairs similar to those in unit testing, or by writing specifications that offer expected outputs for any given input.

Test execution & result inspection Related test cases are usually gathered in a "test suite", where developers have a tool to run a test suite automatically, e.g. JUnit testing framework. As for result inspection, a test oracle is used. The "oracle" can determine from the inputs, expected outputs and actual outputs if the system behavior is as intended.

The limitations of automated tests often make it impossible to gain a full test coverage, due to the large amounts of states, inputs and outputs within a system [51]. Another problem is when critical components of a system must be identified in order to justify the need of automated tests. If any problems occur while executing a test manually, they are usually still present even when the test is automated [44].

Several challenges are introduced when adopting the automation technique. For an efficient use of automated tests, these challenges must already be considered at the design phase of the system. Today, there are development methodologies such as test driven development (TDD) that help. In larger projects that have hundreds, or maybe even up to thousands of automatic tests, a challenge will be to maintain the test suite, especially if there are continuous code changes. To meet this challenge, Meszaros introduced twelve principles to take into consideration when building an automated test, known as the Test Automation Mainfesto (refer to table 2.2) [33].

Automatic tests tend to be static and therefore have problems dealing with mutable variables such as time. One approach to deal with this is by adopting model-based testing, as described in Section 2.1.5.

Table 2.2: The Test Automation Manifesto [33]

Principle	**Rationale**
Concise	As simple as possible
Self checking	Automatic test reports
Repeatable	Test can be re executed at any time, with no extra effort
Robust	Test always produced the same result
Sufficient	Tests verify all the requirements of the software being tested
Necessary	Each test contributes to the specification of desired behaviour
Clear	Easy to understand
Efficient	Executed in a reasonable time
Specific	Failed test points out a specific piece of broken functionality
Independent	Each test can be executed by itself
Maintainable	Easy to modify and extend
Traceable	The test traceable to the code and to requirements

2.1.5 Model-based Testing

Three common problems in system testing are: the lack of time, complex applications and fluid requirements [22]. Manual testing is time-consuming and difficult to conduct, therefore not recommended. Automatic testing on the other hand, is traditionally established by static scripts, and when given enough time and testing, these scripts will cover the system behavior. However, what happens if the system behavior gets modified? The tests will not be accurate anymore and fail. To counter this issue, developers have introduced the idea of model-based testing (MBT). “Model-based testing is a testing technique where the runtime behavior of an implementation under test is checked against predictions made by a formal specification, or model." according to Colin Campbell, Microsoft Research [20]. In other words, a model is created that describes how a system should behave and respond to a given input/action. If we supply an input or action to the model and the system, we can test if they respond in the same way [22].

MBT is usually used as a black-box testing in functional testing but can also be used for quality testing. It has many advantages including a high degree of automation, possibilities to generate high volumes of varying useful tests, as well as to evaluate other types of tests such as unit tests and regression test suites [41].

In studies conducted by Rosaria and Robinson for graphical user interfaces, and Agrawal and Whittaker for embedded control software, MBT has proven to be a very effective test technique, especially in small applications and/or state-rich systems [39, 9].

There are many different types of models. They vary in form and size, e.g. states, sets, grammars, combinations and more. Each one has their own advantages and it is important to choose the correct model type that corresponds to the system being tested. Once the model is constructed, it is translated into a state machine. Several algorithms for

automating this task exist and are presented in [37]. One approach is using an operational mode as suggested by Harry Robinson [38]. In an operational mode we see when certain inputs (user actions) are available and show how the system will react to the inputs in terms of output, which can be modelled as a state machine. However, Robinson also states that "a finite state model used in representing the behaviour of an application is likely to have many, many states – so many that it would be tedious and unrealistic to create and maintain the model by hand". With this in mind, once the state graph is created, the test can proceed to state transition testing. In figure 2.4 are the basic steps of the model-based testing procedure. First the system gets modelled according to a chosen modelling method. Together with a selection (automatic or manual) of system requirement(s) that are interesting to test. The test(s) is then get generated from the model. Lastly, a concretization of the test takes place in order to be able to execute it on the system. After the test is executed we evaluate the result.

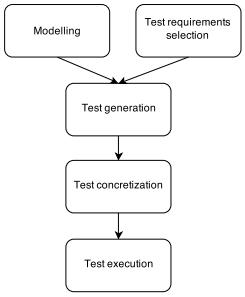

Figure 2.4: The model-based testing steps. [11]

State Transition Testing

In an event-driven environment, there will be a very large or infinite number of scenarios. Together with a finite amount of time, it will be impossible to test everything. In such a case it is crucial to focus the testing parts that are the most important. To get a better overview of the system and what is tested, the system gets translated into a FMS. FMS is a common method used to model all kinds of systems and their behavior. It consists of nodes to describe the state and rules for transitions between the nodes. The model consists of inputs, outputs, start and finish state. From the model, a dynamic test environment can be created and make sure certain states and transitions get tested. A FMS can be converted into a test sequence by applying different graph traversing algorithms; e.g. random walk, shortest paths first, most likely paths first, etc. [24]. Subsequently, a test is now a path of states containing inputs and outputs. The inputs get executed on the system and outputs of the system are checked against the expected results. This type of testing has no built in function for handling timings. However, in this thesis we will use State transition testing and purpose a way to include timings.

Finite state machine

A finite-state machine (FSM) is a mathematical model used to represent a machine made of one or more states for example a computer system. The model consists of a finite number of states, transitions between those states, and actions. This is used to inspect how the logic

runs in a system when certain conditions and actions occur. A FSM begins in a start state, goes through transitions depending on inputs and actions in different states [3]. FSM is a common way to model digital systems and software and therefore often occur in Model-based testing. In this thesis we will use FSM as a tool in order to model the environment of our system. See figure 2.5 for a FSM model of a door.

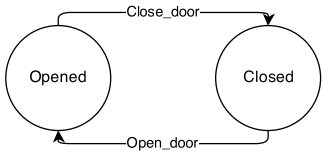

Figure 2.5: Finite state machine model of a door

Difficulties and drawbacks in MBT

MBT provides a lot of benefits but one large drawback most researches agree on is that the development and creation of the model requires a significant amount of resources, time and skills. Studies by El-Far and Whittaker state that “MBT requires sizeable initial effort. Selecting the type of model, partitioning system functionality into multiple parts of a model, and finally building the model are all labor-intensive tasks that can become prohibitive in magnitude without a combination of careful planning, good tools, and expert support." and “MBT demands certain skills of testers. They need to be familiar with the model and its underlying and supporting mathematics and theories" [18].

One of the most important and common methods to model systems is FSM. Unfortunately, some systems, even small ones, consist of more states than a computer can handle, which is the so called "state explosion" problem [48].

Furthermore, if any generated test fails, it has to be decided whether the failure is in the system or the model. The MBT tests are based on the system and therefore an error might be present in both. Thus, it might be more difficult and time-consuming to find the cause of the failure compared to tests not based on the system such as manually testing [18].

2.1.6 Testability

In testing, it is not only different methods and techniques that determine how well a system is tested. The kind of system and its structure also play a large role. Voas and Miller introduce the term testability [50].

Testability - is the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [45].

Since the term got introduced it is more of a standard and many papers have been written about how to achieve a high testability of your system or software. Pressman describes a number of characteristics and quality attributes for a testable system [36, p. 482-483].

Operability The system must be working, few bugs that block the test execution.

Observability Given inputs provide distinct outputs. The system state and variables are visible during the execution. Test failures are easily identified.

Controllability The ability to put a system in a certain state by manipulating its inputs. Every possible output can be generated trough some kind of input.

Decomposability The system is built up by different modules that can be tested separately.

Simplicity The simpler structure and implementation of the system, the easier the testing would be.

Stability Try to avoid or minimize the amount of changes in the systems behavior. Recovers well from a system failure.

Understandability The more we understand, the smarter we cab test. The interaction between internal and external components are well understood. Changes are well documented and communicated.

Having a high degree of testability in a system enables a higher degree of test automation and is therefore an important design goal when developing a system in order to lower the testing costs. For this thesis where an existing system is investigated, analyzing its testability will tell us where and which kind of testing can be automated.

2.2 Markov chains

In the study of probability, it is common to study independent probability processes such as coin flipping or lottery. However, there are many processes whose future states and transitions are determined by its earlier values, a so called Markov process. If the Markov process has a state space that is discrete, i.e. it is finite or the countable space is discrete, it is called a Markov chain. The chain is based on the current state to predict the next coming state and it is used to analyze how a variable or system outcome changes at runtime. For an effective use of Markov chains our model must have a dependency between the states [34]. An example of this would be probability of a stock, which if it has increased in value over one day, will likely increasing in value the next. In contrast, probability of picking the correct lottery numbers one week, knowing the previous ones does not have this property.

Usually, Markov chains are drawn as state graphs, with states and transitions. Each transition is annotated with the probability to take that path. See figure 2.6 for an example chain modelling the weather from one day to another.

Markov chains are commonly used in MBT and go the name Model-based statistical testing (MBST). It is used to create a probabilistic usage profile of the system. The profile will tell us which states and transitions occurs most often in the system and can be used as a complement to direct the testing in one way or another. In this work we will show a way how to convert our model in a probabilistic usage profile and suggest different ways to make use of it.

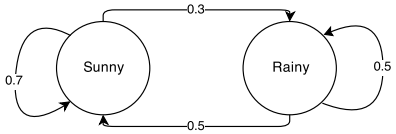

Figure 2.6: Example of a Markov chain for the change of weather the next coming day

2.3 Real-time systems

Real-time systems (RTS) are systems usually made for interacting with the physical world. These systems often interact with an operator or environment through sensors and actuators. An example of an operator would be a user. The system's correctness does not only depend on the computations and result but also has time requirements. A RTS usually has many different inputs and outputs, all with different time requirements. The time requirements are called deadlines and every task within the system has one. There are three types of deadlines: soft, firm and hard. Soft deadlines are not required but are recommended to be met, while hard deadlines must always be met. Firm deadlines can be missed but this can sometimes lead to a system failure [29].

A common way to classify RTS is as Time-triggered or Event-triggered.

Time-triggered systems A time-triggered system runs in a cyclic behavior. Tasks are executed in a predefined order. A scheduler makes sure each task gets their portion of execution time, by sending clock interrupts. Time-triggered systems are deterministic and have a low overhead, but are inflexible [29]. An example of a time-triggered system would be a temperature controller.

Event-triggered systems Event-triggered systems are more flexible in the sense that task are triggered when an event occurs. The system reacts by reconsidering its current schedule. A rescheduling decision is made based on a scheduling policy, current state of the system, resource requirements, and task priorities. The task may be dropped, scheduled for execution sometime in the future, or started immediately by preempting the currently executing task [31]. Event-triggered systems suffer from limited predictability, since we know it will keep the deadlines but cannot determine when exactly and therefore must accommodate for the worst case scenario. This causes event-triggered systems to have a low testability, tests being difficulty to repeat [21]. An example of an event-triggered system would be an access controller, which we will investigate during this thesis.

2.3.1 Real Time System Testing

Even if the testing of a real-time system shares the same goal as regular system testing, this area is less explored and more complex. Especially so when it comes to event-triggered systems because of their lack of predictability. A real time system correctness is based on both the correctness of the outputs and their timeliness. The tester must consider when to the stimulate system, when to expect responses, and how to review the timed event sequence. Further, the test cases must be executed in real-time, i.e., the test execution

system itself becomes a real-time system [23]. Another common problem is that most real-life systems have a huge number of inputs, requiring a larger number of test cases than what is possible to execute in the time allocated for testing [31].

2.4 Physical Access Control System

We look at Physical Access Control System (PACS) in this thesis because it is an event-triggered real-time system and interacts with users. Furthermore it also is time-dependent and the PACS setup and implementation differs depending on the environment. All these properties are hard to test with automatic tests and therefore a good system to try our our model.

The objective of a PACS is to regulate users for accessing resources. It determines the right for a user to access certain areas, such as building entrances, office rooms, warehouses, etc. The system can grant access based on user rights, group rights and schedules. The purpose is to ensure that access is given only to people who are permitted to that location at the given time.

In order to grant access, the system needs to identify a user upon an entry request. For the PACS, there are different ways, but the most common and widely-used one is by individual access cards. An additional safety feature which can be added, would be a compulsory pin code to the entry request. Once the user has been identified, a database look-up will be performed. The database consists of different constraints and restrictions for the users such as schedules, user facility access, and group facility rights. After the look-up is completed, it will confirm the identity and the rights to enter of the user by giving or denying access.

The PACS used in this thesis does not employ a central database; instead, all data is stored locally. Every unit in the system has a local database that is synchronized over a network. The advantage is that every device works independently and can operate even if the network goes down [42].

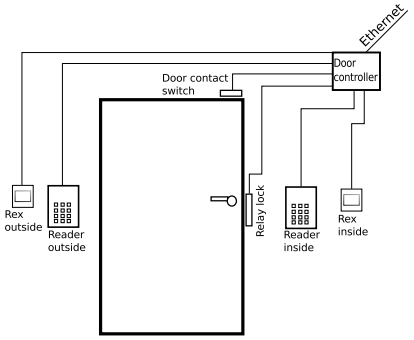

Figure 2.7: The PACS door setup, picture source: [42]

2.4.1 Inputs and Outputs

The PACS supports a variety of components. The components are used by the user to interact with the system in order to gain access. The supported access control components

supporting our system and used in this thesis are: card and pin-code reader, Request-to-Exit (REX)-button, door contact switch and relay lock as described in 2.7

The card and pin code reader is where a user swipes their card and thereafter enters their PIN. The reader then communicates to the PACS over one of the supported protocols, such as Wiegand-26 [5]. The user is then identified and the system determines if access should be given.

In several cases identification is not needed, such as exiting a building or a room. In this cases it is sufficient to have unlock-button a so called Request-to Exit (REX). A REX is used to grant access regardless of the user access rights, but can be scheduled and is commonly placed inside the building.

The door contact switch consists of two magnets mounted in the door and the door frame to determine the doors state. The relay lock is a lock that can be controlled by regulating its power supply to remotely control the lock state.

2.4.2 Stimuli and Logging System

In almost every type of security system logging is an important aspect. Even if all systems try to work proactive, it is not always enough. By collecting and analyzing logs, you can track down abnormal behavior or certain usage of the system. The PACS, being an event based system, uses a logging system built on events. Events will be created based on time, stimulus or administration changes. Stimulus is different types of interaction (inputs) users have provided the system. In this thesis we will mainly focus on the stimuli based and a few timed events.

When one of the access components gets stimuli, one or many events are created and stored in a system log. Each event has a time stamp and additional data about the event (could for example be identification data). Table 2.3 lists the different types of inputs, that can occur within each component and would create at least one event.

The timed events we will also use are so called time-out events. Those occur when the system times-out; for example: if a door is open too long or a user has swiped their card but takes too long to enter their PIN. In the system used in this thesis logs are accessible through a few different ways. For a system administrator, the web interface would be most handy. However, in our case we will focus on automation and then a text based version would be suitable, which also are accessible through SSH/telnet.

Table 2.3: User inputs

Component	Input
Card-Reader	swipe valid card
	swipe invalid card
PIN-Reader	correct pin
	incorrect pin
REX	press REX
Lock	lock
	unlock
Door	open
	close

Chapter 3
Related work

There has been a large amount of research in the areas of model-based testing and test generation. However, the application of these techniques to real-time systems has not been explored nearly as well. There are a number of methods and tools aimed at automatic test generation. In this chapter, a handful of them will be presented to illustrate how they were used to solve similar problems.

UPPAAL

UPPAAL is a tool for modeling, validation and verification of real-time systems [30]. It has been developed by Uppsala University, Sweden and Aalborg University, Denmark. The tool models the system as a network of timed automata that runs in parallel. A timed automaton is a finite-state machine extended with clock variables. The state of the system is determined by the locations of all automata, the clock values, and the values of the discrete variables. Every time one of the mentioned values changes, all automata will then synchronize and a transition might take place which leads to new state. UPPAAL has successfully been used for modelling audio protocols, web services etc. [14]. However, it has also been used for test case generation for real time systems by tracing around the visible states in the automata which is then converted to a test case [23].

Learning-based testing

As a result of more data becoming available, Machine Learning is being used more frequently within computer science. Machine learning (ML) is a sub-domain within Artificial intelligence (AI). In general, ML uses training data to create a model which can be used for either classifying unclassified data or creating new data. A further development in testing and for Model-based Testing is to extend it with ML, in a form of learning-based testing (LBT). LBT was introduced by Meinke and Niu and is based on three components [32]:

1. a (black-box) system under test (SUT)

2. a formal requirements specification

3. a learned model of the system

The difference from model-based testing is that in LBT, the model receives feedback from the generated tests and will keep modifying the model until a failure is discovered or a decision has been made to terminate testing. LBT has the ability to do test case generation, execution and evaluation (oracle). The aim of this approach is to automatically generate a large number of high quality test cases.

GUI testing

Graphical user interface (GUI) are usually event-driven i.e. if a button is clicked an event is released and therefore share the same problems as an event-triggered RTS in terms of testing. Both have an operator (user) which on a given input should generate a output [40]. There are three common techniques for GUI testing:

Capture and replay is when user interaction (mouse and keyboard events) get recorded when the GUI is used, which in a later case can be replayed. This can be used to test if the GUI reacts identical as when the test was recorded. The drawback of with this technique is if the GUI changes behavior or appearance, old test will be outdated and new one has to be created [10].

Event-triggered In a GUI, you can exploit the fact that all components structure and function is well known. The tester uses a "robot" in order to identify and executes components showing up in the GUI and in this way simulate usage of GUI. In a test you decide on beforehand which component(s) that should be tested and is therefore more flexible the "capture and replay" since the test is predefined. The tests will still work if the GUI changes appearance or behavior [40].

Model-based uses abstract model(s) of the GUI in order to model its behavior. These behaviors can be found by studying different event-flows that occur in the GUI. The model can then be used to generate automatic test for the GUI. It would be feasible to automatic create model based on the GUI. However, this is usually not possible therefore the model is created in advance [52].

Fuzzy Testing

Fuzzy testing or providing fuzziness to a system is a testing technique for finding vulnerabilities by generating invalid, unexpected or random data for inputs to the system. It aims to discover system failures by testing extreme cases, that are often missed or forgotten during development. By failure we mean system crash and unexpected behavior such as infinite loops, memory leaks etc. [27].

Fuzzy testing can take place as both black-box and white-box testing and there are two different ways to create fuzzy input: Mutation-based and Generation-based.

Mutation-based The input is generated from either deterministic or probabilistic models. This requires already known valid inputs, but can often generate very accurate and

extreme inputs from the model and is the most common approach for fuzziness. An example would be to send in a negative number to a function that calculates the area of a square.

Generation-based If no available inputs are accessible, the inputs can be generated from a set of rules describing the inputs syntax. It requires a good understanding of the specification to be efficient.

Fuzzy testing has been discussed and used widely in the industry and is considered one of more effective techniques for discovering bugs [49]. There are many different approaches and methods to make the fuzziness more effective and comparison between the different methods can be seen in [19]. Brian S. Park suggests a combination of symbolic testing and fuzzy testing in so called hybrid testing. Fuzzy testing has its weakness in coverage measurements but can efficiently explore a program's state space. In contrast to symbolic testing, which tests a program by treating the program's input as symbols and interpreting the program over such symbolic inputs, can effectively produce a high coverage. Hybrid fuzzing has been showed to be more efficient in time and space compared to other testing approaches [35].

Test generation

In order to do extensive testing of a system, the test process must be automated [26]. An important and critical part of automatic tests is the generation of inputs that will get executed on the system. Often certain values are more likely generate an error and are therefore important to test. Boundary value analysis is the theory for finding those values or partitions in the input space. There exist several to validating predefined boundaries [28]. However, these methods do not provide a way for calculate these boundaries, which is one of the problems with timings in this thesis. A common way to for discovering theses boundaries is by using a random mechanism to explore the inputs space. But with a large inputs space random exploring often result in many meaningless test. In contrast to random exploring, [12] suggest to derive the boundaries by observing the input-output relation for a real-time system.

When tests are generated for a system, it is important to have a test adequacy criteria. It provides a measurement of test quality and can be used for guiding the test generation [53]. From a model based testing view, some common coverage criteria are [47]:

State coverage is the percentage of states covered by tests.

Transition coverage is the percentage of transitions covered by tests. Where the transitions are the edges between the states.

Path coverage is the percentage of paths that is covered by tests. By covering all paths it corresponds to exhaustive testing. A path is connected states and transitions in between a starting state and ending state in the graph.

Other types of coverage measurements are grammar-based and specification-based.

Real-time systems often communicate with an environment in terms of inputs and time and from that produce a desired output. Developers usually specify these systems with scenarios. A regular real-time system can have up to hundreds of thousands different scenarios

and each one would require at least one test case. [46] suggests a so called Verification patterns (VP) to address this problem. In VP the system scenarios get categorized and classified into patterns. For each pattern a test template is made that can be used to test all scenarios within a pattern with a low cost. The problem with many scenarios occur in our thesis as well, since our system under test is a real-time system.

Chapter 4
Experiments

In this chapter we present the experiments we carried out for two model-based test generation approaches, a manual and an automatic built. Its purpose is to show how this can be done on a real system and how well such an approach works. A comparison and motivation for both approaches is presented, in order to to show their properties for testing a system. As the system under test we employ a physical access control system.

The chapter starts by describing the different parts in the experimental setup, their purpose and their configuration. Once all the components are in place we need to connect them, therefore next section describes the connections such as protocols etc.

Section 4.4, is more theoretical and starts to explore the method to create a model for our system, how this is used for testing. In this section, examples of the model and test generation will take place, starting by creating a model, then use the model to create tests and see the correctness. Lastly, the experiments will be presented in Section 4.5.

4.1 Objective and Motivation

Creating automatic tests for an event-triggered RTS is a difficult task, because of automatic test limitations in testing different timings. For a system like PACS, the timings will vary depending on its setup, devices etc. Therefore it would be advantageous to have tests that could adopt and change depending on the systems appearance. In order to make the automatic testing less demanding (maintenance and effort), we want to adopt a technique a that generate few dynamic tests for testing timings requirements. By dynamic we mean changeable timings within a test case to gain more fuzziness in the testing and therefore be more likely to find errors. For a system that interacts with an operator such as a human, it would be advantageous to recreate the interaction in form of delays and inputs. It could be used to prove for the developers or customers that the system is working in a correct way but also to "replay" erroneous interactions in order to track down and fix the bug(s).

In order to get all these properties into a "test suite" a model-based approach was cho-

sen. From the model, automatic tests can be generated, but it can also be used as a test oracle for test evaluation.

The objective for this experiment is to show how a "test suite" could look like, how it would be used on actual system and how well it fulfil the requirement.

4.2 Setup

In the experiments there are three main hardware components: The PACS, an Arduino and a desktop computer see figure 4.2. Here comes a motivation for and description of each one of them.

4.2.1 PACS

The PACS work as described in section 2.4. It has an Ethernet connection for connecting to a network. Over the network it can communicate with other PACS or components. It has the configuration of one door with a lock and door monitor. A user can identify himself by swiping a card and enter their PIN. For exit a REX button is placed, which a user can press at any time to unlock the door. We have chosen this configuration to involve as many components as possible in the experiment.

The I/O to PACS are electrical signals, which are used to communicate to readers, buttons and doors. It can also be used to read the state of the door lock. See figure 4.1

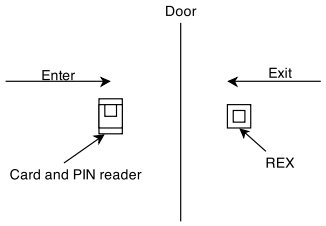

Figure 4.1: The PACS configuration

4.2.2 Arduino

In order to have an automatic model that generates tests, it would be impractical to manually enter pin codes, swipe cards or open the door. By using hardware for stimuli, we can have tests created from the model continuously. All we need is a device that can generate the desired stimuli and have an Ethernet connection since it is required for commuicating between the components. For example, we could use a computer with an I/O card, an Arduino Mega 2560 card with an Ethernet shield or to use a Raspberry Pi with Ethernet (model B). A shield is an extra component that can be connected to the Arduino in order to extend its capabilities for example with an Ethernet port [1].

One important requirement for the device is to be able to send signals with the voltage of 0..5V, since that is what the PACS uses. This is no problem for the Arduino card since its ports operate on 5V [2] and for the I/O card there are 5V ones. Raspberry Pi however, only operates in 3.3V [7] and would require a step-up converter in order to provide 5V.

It would be advantageous if the device be attached close to an actual system in use, since we need to connect the device to the systems in/out ports which makes the Arduino and also is the chosen as our stimuli device, because of its small size and good handling of accurate timings.

4.2.3 The computer

The main purpose of the desktop computer is to build the model as a graph. This is done by retrieving information from the PACS event-log. From the model it can create, execute and evaluate generated tests. The whole model generation is written in Python, but any object oriented language could have been used for this task.

An advantage of using a computer for storing the model and test generation is if the testing takes place in a larger scale with multiple Arduinos, only one computer would be needed that generates instructions for all the Arduino's.

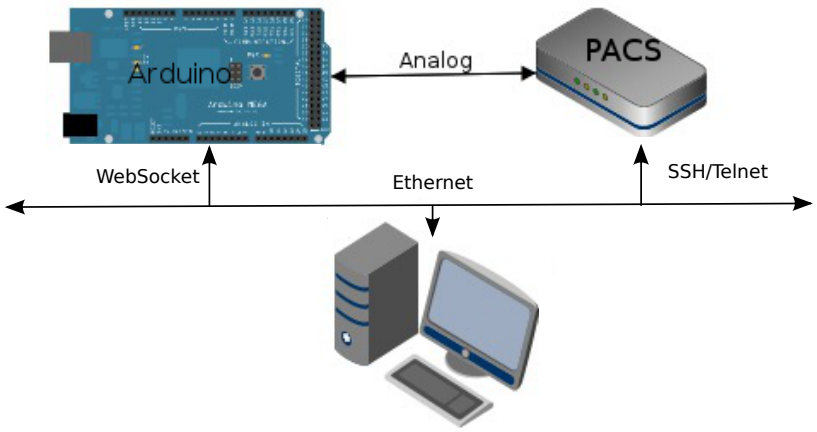

Figure 4.2: Experiment system setup

4.3 Connections and Protocols

As mentioned earlier, the setup is based on three components (PACS, Arduino and a desktop computer). They are all connected to a network by Ethernet connections, which is reliable and all devices support. Other types of connections that exist in the setup is USB for Arduino used for debugging and analog electric connection for sending stimulus to PACS. In this section will go through how the devices communicate and the protocols they use.

4.3.1 Arduino and PACS

Ardunio offers both analog and digital electric in and outputs. These can be connected to the PACS for generating a desired input, that is identical the components stimulus, for example swiping a card.

4.3.2 Computer to PACS

Through the Ethernet connection the PACS web interface can be accessed, where one can edit the configurations, user and user group rights, read the event-log etc. A more convenient way for us to connect is to connect through SSH/Telnet. In here, we can access the same information and configuration as the web interface, but from command-line, which is more effective for automation and therefore used in this thesis.

4.3.3 Computer and Arduino

The Arduino and computer communicate over a WebSocket. A WebSocket connection is established over the Ethernet network. The WebSocket is used to send JSON formatted strings, which the Arduino is programmed to read (if formatted in a correct way) and will execute the directions from the string. A direction would in this case be a stimulus together with a timing which defines the desired delay. Multiple directions executed in sequence are viewed as a generated test case. This type of connection requires an Arduino with Ethernet extension (shield). The standard way of communication for an Arduino is through Serial/USB and could have been used for sending the JSON-strings, but is slower and would need extra configuration. By using Ethernet it allows us to controll multiple Arduinos from one computer as long they are under the same network. In our case, the Serial/USB is mainly used for debugging purposes, for instance to examine the Arduino current and next executed stimulus.

WebSocket

WebSockets provide a protocol between client and server which runs over a persistent TCP connection. The client is typical a web browser but can be in other forms too. Through this open connection, bi-directional, full-duplex messages can be sent between the single TCP socket connection (simultaneously or back and forth from A to B).

JSON

JavaScript Object Notation (JSON) is a portable data interchange format. While its structure is recognized natively by JavaScript, its formatting conventions are easily recognized by other C-like languages. JSON.org has an excellent description:

"JSON is built on two structures:

A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures." [6]

4.3.4 Time Synchronization

Time synchronization is an important task and common problem in computer science since it is necessary in order to synchronize data from different devices. Usually clocks drift or get set wrongly. You want then be able to synchronize the clocks in a simple way with as little delay as possible. In our setup we have three components which we want to share the same clock, since the PACS configuration might change depending on the time. There are plenty of solutions and algorithms out there like Network Time Protocol (NTP), Precision Time Protocol (PTP) and Global Positioning System (GPS) some better than others depending on your objective. In our case we need a solution that gives an error with at max a few milliseconds, otherwise the order of some events might be changed. Since already the PACS have supports NTP we will be using that to get a good solution.

4.4 Model Creation

We have looked at two different ways for creation of the model. The first one is a manual approach, where the graph is simply drawn by hand. The second one is an automatic method where an Event-flow graph (EFG) is generated based on the systems event-log. Either way both graphs consist of states, where states represents a stimulus for example swipe card, lock door, open door etc. and the edges describes the transitions to the next possible stimulus or output.

4.4.1 Manual Model Creation

For an access system, states and edges are well defined and are limited in number and therefore drawing a general model-graph manually for such system is quite an easy task. Figure 4.3 is an example of a manually drawn graph for an access system. The graph is very general and could represent basically any access system. A general model can be used for different types of the same kind of systems, but problems to model specific parts for one system.

Since the problem is about "mimicking" user behavior, the model is based on usage of the system and the outputs that will generate. The states get arranged and connected in logical order. For example, the state "open the door" does not appear before the door is in unlocked state.With a complete model, a test get generated by finding a path in the graph, the states in the path gets translated and generated into a JSON-file.In the file the Arduino can read the external stimuli and their order it should get executed. To check if the correct outputs were made from the system has to be done manually.

The higher complexity a system has, the more difficult manual creation will be. A complex system usually involve many states and it requires a high knowledge of the system, to manually create a model of it. The main problem with a manually created graph is that, there is no good way test your model correctness i.e. how well does it model the system. A more common approach is to model small pieces of a complex system and then try combining them. But if the system suffers from a large state-space so called "state explosion", this is impossible. Another drawback is the lack of ability to adopt changes in the system, for example if the REX-button would be replaced with a card-reader, the

whole model will have to be remade. However, the advantage is that it can be produced and used early in the development process.

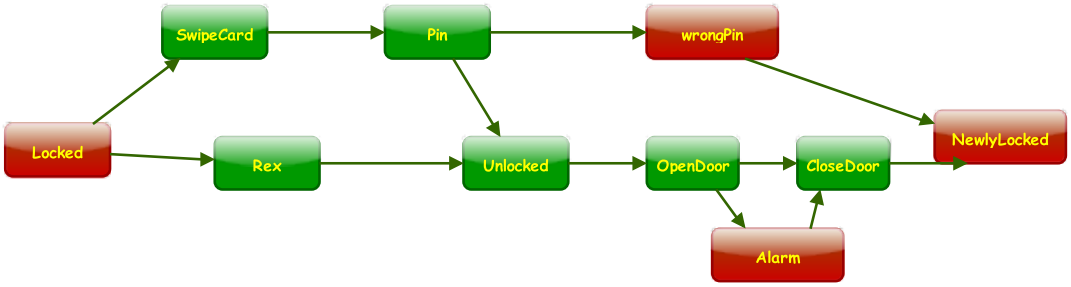

Figure 4.3: Manually created model for PACS

4.4.2 Automatic Model Creation

This section describes the techniques used to construct the event-flow model for the PACS. It contains three parts: (1) algorithms to create event-flow graphs; (2) a technique to generate tests from the event-flow graphs; (3) execute tests on the system and verify them against the model automatically. We follow the notation mentioned in figure 2.4.

4.4.3 Algorithm to Create EFG

The construction of the event-flow graphs is based on the identification and classification of events and their relationship. The classification is used to determine if an event is a stimulus or not. For example a CardPresented is an event with stimulus when a user swipe a card, but the AccessGranted is not. Rules for classification are setup in advance. Table 4.1 describes the events created within the PACS and their classification.

Events are collected over a SSH/Telnet connection from the PACS to the computer. The events get filtered based on a given time-frame and relevance. Relevant events are those we either can generate by stimulus from the Arduino or get an output as a result of the inputs. An irrelevant event is for example user database changes, schedule events etc. Filtering is a crucial task, since if events that are outside our scope get in the model, the model will look for them during the test phase and the test will fail.

The computer will then have access to a list of relevant events within a given time-frame. Events are sorted based on their time-stamp. The Event-flow graph (EFG) defines an event to have a relation to the closest event in time and connect them by a directed edge. In figure 4.4 is an example of this process with a small event-log from the PACS.

In the PACS event-log the same type of events will occur multiple times, but in the graph each event only exist once. To prevent the graph to become cyclic, one or many start or end-state(s) are selected. From an end-state no exiting edges are added, in other words once reached it cannot be left. For the PACS we have defined when the door closed and locked, or a user is denied access as end-states, in other words following events:

- **AccessControl/AccessNotTaken/** - Access was given but not taken, the door is locked.

- **AccessControl/AccessTaken/ followed by Closed/Door/DoorPhysicalState/State** - Access was given and taken, the door is now closed and locked.
- **AccessControl/Authentication/Denied/** - Access was not given, due to a wrongly entered PIN code.
- **AccessControl/CredentialNotFound/Denied** - Acess was not given, due to card not found in database.

An example of the EFG creation is in figure 4.4. First a user swipes a card that does not exist in the database, for example an expired card. Three events are created Activity/CardPresented/IdPoint/, Access/IdData/IdPoint/Request/ and AccessControl/CredentialNotFound/Denied/. Since AccessControl/CredentialNotFound/Denied/ is a predefined end-state no relation is created to the next event. Now, the user (or another) swipes a correct card and enter a correct pin, access is granted and the door becomes unlocked and the user goes through it. In the end of the event sequence we find the events AccessControl/AccessTaken/ and Closed/Door/DoorPhysicalState/State/ which together is defined as an end-state.

Table 4.1: Relevant events in PACS

Event	Description	Stimulus
Activity/CardPresented/IdPoint/	User swipe card	Yes
Access/IdData/IdPoint/Request/	Card identity database checkup	No
AccessControl/CredentialNotFound/Denied/	Identity no found	No
Activity/CodeEntered/IdPoint/	User enter PIN	Yes
Access/IdPoint/PIN/Request/	PIN checkup	Yes
AccessControl/Authentication/Denied/	PIN not correct	No
Activity/IdPoint/REX/	REX button pressed	Yes
Access/IdPoint/REX/Request/	REX button checkup	No
AccessControl/AccessGranted/	Access granted	No
Accessed/Door/DoorMode/State/	Door unlocked	No
AccessControl/AccessNotTaken/	Door locked, never was opened	No
Door/DoorPhysicalState/Open/State/	Open door	Yes
AccessControl/AccessTaken/	Access taken, lock door	No
Closed/Door/DoorPhysicalState/State/	Closing door	Yes

Delay

The SUT is a real-time system and therefore it is time dependent. Therefore it is important to include time in the model. This is done by attaching a list of timings for each edge in the EFG. The reason we use a list of timings is that it enables us to choose a mathematical model such as MAX/MIN, normal-distribution or others that suits the preferred test generation best.

What separates two users usage of the system is:

1. Kinds of inputs and order of them

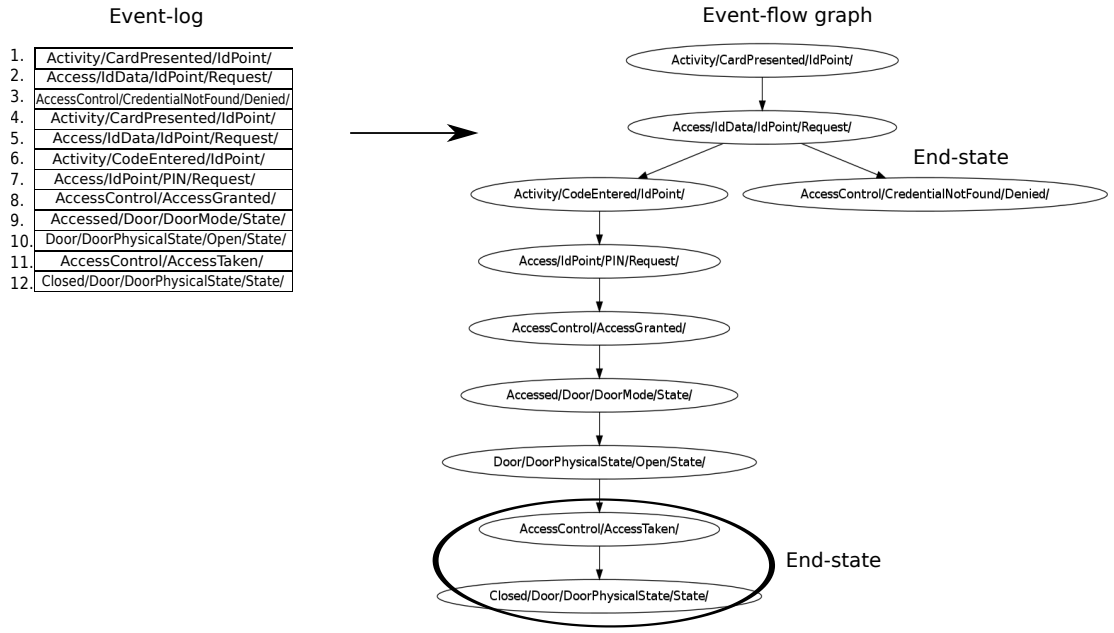

Figure 4.4: Example of the creation of EFG from an event-log.

2. Expected output

3. Timings between different inputs

The model we have built so far takes care of the first and second item. We now describe the third.

As previously mentioned, each event and edge exist only once in the EFG. However for every time an edge occurs the timing between those events gets calculated and added to a list in the edge. From the event-log we can retrieve more information asent events and their relations. Each event has a time stamp and by subtracting two time stamps we get the time the duration between two events. Doing this for all related events according to our model i.e. all events with an edge in between, we get the accurate timings. Figure 4.5 is an example of this. For each time two events occur next to each other timewise, a timing is added to their edge time list, for example $Activity/CardPresented/IdPoint/ \rightarrow Access/IdData/IdPoint/Request/$.

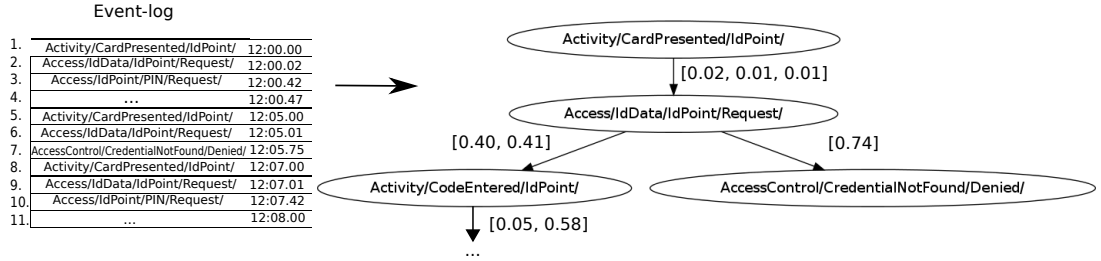

Figure 4.5: Example of the creation of EFG with list of delays attracted the edges.

Markov Chains

The model can now describe the different input orders, expected output and the timings between them. We now have everything we need for representing one use, but what about many?

Earlier we mentioned that in a time-dependent system with a high amount of inputs and states, it is impossible to test every single combination of inputs and timings. In such a situation you usually want to focus towards the most common inputs orders and timings combinations, so these are guaranteed to be working. For example, lets say 10% of all users enter their PIN wrongly. We want the tests to be the **most likely use**. Then the test should 1/10 of the times enter a wrong PIN and 9/10 enter a correct one. With a completely random path selection both states would have equal chance. Instead we can use the size of the timings list to see how many times a specific edge was passed. If a state has more than one exit edge we can sum the sizes of lists and easily determine a probability for selecting a specific edge. In this way we can generate a Markov chain from the EFG, that can be used as probabilistic usage profile. See figure 4.6.

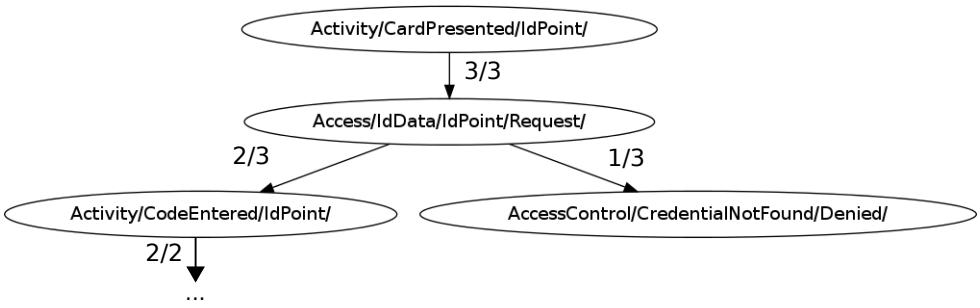

Figure 4.6: Makrov chain calculated from the EFG in figure 4.5

4.4.4 Test generation

The complete model consists of inputs and outputs in form of states, timings and probability in form of list linked to edges, all needed for generate usage scenarios. The next step is to do this generation, by generate tests from the model. This is done by traversing a path in the graph. The path is based on the Markov chain of EFG, and is a sequence of events and generated timings. The event sequence gets screened for events that contain external stimulus. Those events together with generated timings get translated into a JSON file and sent to the Arduino for execution. In order to be able to reproduce the test case, the JSON-file gets stored in the computer.

4.4.5 Test Oracle

Both models can be used as a test oracle. They can based on a generated input predict the systems output. The predicted result is used as an oracle by being compared and evaluated to the event sequence created by the test. However, the EFG will be a more complete and accurate oracle, since it can determine which output(s) and events will be created given certain stimulus with delay.

4.4.6 Test Execution and Verification

The created JSON-file gets uploaded to Arduino. In the file the desired stimulus together with given timings, which the Arduino can read and execute thereafter. Since the Arduino provides analog signals to the PACS and timings based on earlier usage, it will look an actual user access the system..

Once the Arduino signals that it is done executing the test, the computer now fetches the events that were created from the test. These events then get filtered and sorted in an identical way as when the model was built. The path with timings can be used as a test oracle.

For the evaluation and test verification we compare the oracles predicted result (the path) with events generated in the system from the test. Fist we control the events, if the path and test event sequence matches in events and their order. Then we need to check if the generated timings occurred in the system. The timings may not be exact because of for example execution order, high load in the PACS etc. To deal with this we accept a margin of error $+-1-3\%$ for the timings. If both pass, the test is considered to be passed, otherwise it failed.

4.5 Results

It is critical to have an extended knowledge of the system and the requirements for testing in order to choose a good model, and in most situations, a combination of different modelling methods is needed. This experiment had described two models; one manually built, and one based on EFG.

We have reviewed them based of following criteria:

Automation and Effort

One important aspect within this thesis has been the use of automation. We want to change the way testing usage of the system, but still do it automatically. Since MBT has a high degree of automation, the property of automation will not be affected. It allows the testing framework to achieve a fully automated test case generation, test execution and test evaluation based on the model. For both models, test generation and execution require no effort. For evaluation purposes, the EFG has a better process to produce expected results as compared to a test oracle. This is done by comparing the generated with the actual event path.

The manually created model does not have the ability of a complete oracle. A path can be used as a predicted result, but it cannot compare and evaluate these without manual help, by linking the states are with corresponding events manually.

Automating the building of the model would save resources and make things easier for the tester and there is a difference between the two models in this sense. None of them are fully automatic, the EFG only requires start and end states to be chosen and stimuli events pointed out. This is a challenging task and requires a lot of knowledge of the system, but is nevertheless still better than creating the whole model itself, in terms of effort. In contrast to the manual model that has to be made fully by hand.

The model creating has the time complexity of O(n) since it will visit every event in the log once, which also figure 4.7 confirms. The required time for building the model is linear dependent on number of events in the event-log. In the time we have included, the time for fetching the events from the event-log which is the reason that 1000 events are not five times shorter in time than 5000. In PACS 30000 is the maximum number of events that can be stored in the log and therefore the maximum we have tested. However, it is possible to build the model one more events by having the model fetching events in real-time.

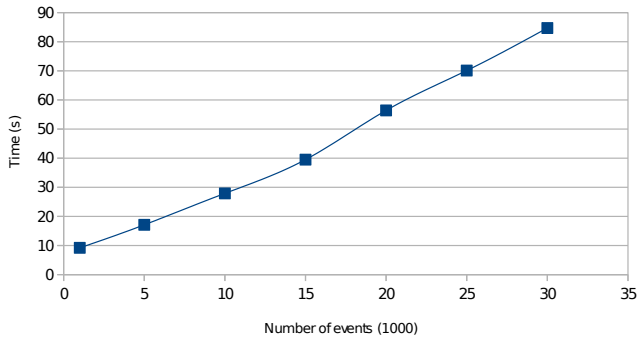

Figure 4.7: Time to build a model based on number of events

Testing

The goals of testing is to prove to developers or customers that the system fulfills the requirements, as well as to discover any erratic behaviors. Therefore, it is important to review the kind of test that is being generated for the system. All testing takes place on a event-based level and are therefore not available to test on any lower levels. The reason for this is to recreate or be as close to the user interaction as possible, which is something the test does very well. Both models are based on a event-level and therefore can not test a lower level. Lower level bugs can still come visible during testing but will be hard to track down without any help from other testing tools.

Generally, the tests consist of sequences of inputs with a delay in between. The order of the inputs is designed to mimic a user's behavior as much as possible. For example, the door will not be opened before it is unlocked. The EFG will create tests based only on sequences found in the event log, whereas the manual model will be more generic. In terms of automation, the Test Automation Manifesto (table 2.2) should be fulfilled.

The EFG can be used as a complete and accurate test oracle, than the manual model. It can generate an ordered sequence of events with inputs and outputs, which can be easily compared to the actual results. See the test oracle characteristics in table 2.1.

Delays

Timings are important for real-time systems and therefore important to test. A real-time system's performance is dependent on the time and the goal of this thesis was to find a way to test this automatically. For a system like PACS, the timings will vary depending how

and where it was implemented, such as the type of devices and distance between them. Therefore, it would not be possible to guess good values, as a baseline is needed to derive them from. Such baseline is non-existent for the manual model, and therefore, either a constant or random delay similar to those found in regular automatic tests would have to be used. However, the EFG has access to timings between inputs from the event-log and with that, will be able to produce very accurate timings.

Creation of pre-recorded scenarios

In the event an error occurs during usage, it would be interesting for the testers to recreate the scenario in order to find the bug. With proper inputs and timings, the model can produce tests that would 'mimic' a human user, something that exists in the EFG. Since the EFG is based on event-log, it can easily recreate scenarios, as long they exist in the log.

4.5.1 Summary

Overall, there are many advantages of using model-based testing for an event-driven system. It can create tests that copy an earlier usage of the system. It is likely that an EFG is superior to a manually built model, which also is true from an automation perspective, even if it is not fully automated. One large drawback is that the system has to be used before the tests can be created. The main reason for the need of timings is to be able to calculate new ones. The problem is that a usage usually will not take place until late within the development cycle. An idea would be to have a manually built model until the tester reaches that point. Overall, this form of testing has a large amount of potential and would be a great complement for testing, especially in event-triggered system for verifying certain qualities and behavior which are time-dependent. The model creation only has to be done wise and is fast (O(n)) which allows us to create models from a high amount of events.

Once the model was built based on the event-log, automatic tests can be carried out that can mirror or be very similar to the usage of the PACS. The system will be more tested on paths (event sequences) that occur more often in the system and fully supports changing timings dynamically between test runs. The manual model there is no built in support for timings and probability, however it could be specified manually but it requires a lot of work and hard to determine good values.

Chapter 5
Analysis

In this chapter the method and results from the experiments, described in chapter 4, will be further discussed in terms of drawbacks, improvements, alternative solutions etc. divided into five topics: Hardware and setup, The Model, Delays, Automation and testing.

5.1 Hardware and Setup

In our setup, we have three components; the PACS, an Arduino and a computer. The PACS, which is our system under test (SUT), is configured with an entrance reader that requires a card and PIN number for identification, a door and a REX for exit. This configuration involves all types of stimulus and is therefore good for the experiment. In the context of a larger scale of testing, multiple PACS would exist and work together. Test setup handles this well, since one computer can be used for multiple Arduinos as long they are connected to the same network.

The Arduino which is only a running loop, has great support for timings and for creating delays, with already implemented functions pausing the execution and a time-resolution of 2 μs which is a great advatange compared to other devices for example Rashberry Pi. An interesting future progress would be to unite the computer and the stimuli device into one. This is likely possible by utilizing a BeagleBone [4], which is capable of reporting back the test results. Nevertheless, a regular computer would still be needed, in order to be in control of multiple access points, which is a big advantage in a larger system. The tests were written in Python but could have been written in any programming language. Protocol-wise, there is no need for change as both WebSocket and JSON are simple and solve the assignment. However, an issue with the Arduino is its limited memory. The testing model had produced a test case with many inputs which in turn delayed the JSON file and may become too large in space for the Arduino. It was not likely to happen in our tests due to the limited numbers of inputs, but may occur if the methods are adopted to another system.

5.2 The Model

In order to "mimic" a user, the solution is a form of 'Model-based testing' (MBT), or in other words, Model-based statistical testing. In MBT, usually the system is modeled. However, in this case, we try to model it from a usage perspective, i.e. how the system is used and what events that have been created, in order to "mimic" a usage in an optimal way. The PACS is an event-triggered system and usage creates events. It is advantageously to base the model at an event level. This will make the model to have an abstraction close the usage and therefore will be more clear and maintainable. It could be interesting, for example, to test the interconnections and communication within the PACS, such as Wiegand-26 [5] but that would require its own analysis to obtain a suitable model which falls out of the scope of this thesis.

A strong advantage with MBT is its high degree of automation where test cases can be automatically generated from the model. For the PACS, the model was decided to be built as an EFG, a common method for modelling GUIs as it is capable to model events and their relations. A relation is created to the following event in the sequence until an end-state is reached. For other modelling methods such FSM, an issue exists in determining how and which states should be connected. Another concern in FSM is the risk of state-explosions. In the scope of this test, since each event only occurs once in the graph and the number of unique events are limited, the problem is highly unlikely to occur. With that being said, if the developers decide to change the system or add many new events, this could happen.

To be able to model timings, a list of timings is added for each edge. The lists can be used for any preferred mathematical model to produce good timings such as MAX/MIN, normal distribution, an average value, etc. By storing each and every timing found, no data is lost, although this implies that a larger memory storage. Nevertheless, as this takes place on the computer, memory space should not be a problem. However, depending on which mathematical model is used, time is needed to create edges with large lists, but is only built once. It is something that should be taken into consideration if something such as a BeagleBone is being used.

There could be situations when two users are interacting with the system at the same time; such as when someone is trying to enter the building and typing their PIN while another person is going to exit and pushes the REX-button. The REX-event and PIN-event would then have an edge in the graph and then be concluded within the same test. This does not present any issues and can be tested by the model. However, another problem that is not handled as well is a cyclic path in the graph. In the JSON structure, each stimulus is designated a timing and whether or not it is going to be used. If a cycle would occur in the path, the same stimulus might occur twice with different delays. To solve this problem, the events have been limited to only occur in the path at most once. Another solution would be to change the JSON-structure to allow multiples of the same stimulus. By doing so, more fuzziness could be tested; for example, by pushing the REX five times but it would require a higher degree of the Markov chain to get a good probabilistic usage profile of the system.

For the creating a probabilistic usage profile, Markov chains are used. From the time lists, the probability can be calculated and used to create the EFG into a Markov chain of the first degree. If a higher degree is needed, the probability should be calculated by creating another graph. Having a Markov chain of a higher degree could be interesting if

previous events are dependent for probability for the next event. The higher the degree, the more past events are taken into account. An example to consider would be, someone who uses the REX is less likely to use the door as compared to someone using the reader. In this test model, the Markov chain of first order is used, as an higher order would be more complex and without any significant effect.

There are many advantages of adopting MBST, the main advantage being that the system test for timings are easier to maintain and keep track of what have been tested. This will increase the fault detection because it allows the developers focus more resources on the actual testing and can in a way, better determine where more testing is needed. If the system is updated with for example, new events, the model will detect this and update itself. Since the model is based on the usage, it is a good way to test an event-driven system which otherwise is difficult to test.

One of the biggest challenges with MBT is the correctness of the model. This signifies that it will be difficult to determine what causes a test to fail, whether it is an error in the system or an error in the model. This challenge is attempted to be solved by following the principles of the Test Automation Manifesto (table 2.2). Both the model and system produce a list of events that are compared against each other. In the test reports, both lists are presented and whether anything is missing in any of the lists, it is then up to the developer to determine the correctness of the test. It is also difficult to do any kind of metrics such as test or code coverage. Instead, the focus should be on requirements coverage in MBT. The model only focuses on user interactions and functional testing, and therefore should only work as a complement to other types of testing such as a unit test.

With a complexity of O(n) the model building is fast and sufficient. In PACS 30000 is the maximum number of events that can be stored in the log and therefore the maximum we have tested. However, it is possible to build the model one more events by having the model fetching events in real-time. The number of events needed to create a complete graph is harder to determine, it depends much how the system is used. It is obvious that the smallest number need is equal the number of unique events that can occur in the system. One idea is to have a list of all unique events and check them of one by one when they occur in the event-log and once every is checked off then model is considered done.

5.3 Delays

An important aspect in automatic testing for this thesis has been timings. The system is time dependent and interacts with users, where every interaction has its own unique combination of stimuli and timings. The goal is to 'mimic' these combinations as accurate as possible. When it comes to timings and delays there are three basic approaches: Constant, random or generated.

The generated timings from our model cannot be completely random. Depending on the environment and users a certain range timings will be more interesting form a testing perspective. However, within those ranges we still want some kind of fuzziness in order to get a better test coverage. The challenge is to find a way that makes sure those timings get tested, but still have enough dynamic tests to cover as many scenarios as possible.

Constant delay

Constant delay is the simplest way to create delay and therefore the most commonly used in automatic testing. It can either be a value assigned beforehand or manually changed every time before executing. This type of delay can be used in the manually created model. It can also of be generated from the EFG-model, but it will decrease the coverage of the testing and not be able to 'mimic' the users. Constant delay is a good way to test the system for handling any kind of delay, for example in the beginning of the development process of a system.

Generated delay

Form a real-time perspective, it is desirable to test different timings since the systems correctness depends on it. One way to do this would be to mark the places where delay occur, and assign them a random value. For real-time systems this would be a better approach than constant delay in terms of testing the system correctness.

One problem with random delay is the recreation of tests. For example after a test fails, you do some changes and then you run it again to see if the changes fixed the error. If we try to rerun the test by doing a new test, new timings would be generated and therefore not the same from a real-time perspective. To solve this problem a combination of constant and random delay can be used. First you need to store the random delay, then be able change the constant delays to this value during a rerun. In the model the produced timings are stored in a JSON-file, that we are able to upload again to rerun the test.

Usually when testing delays, some timings are more interesting than others for example values around deadlines, most common occurred etc. In other words we want to test delays around certain values but still have a property for fuzziness to get better tests. If one knows these values in advance, they can be included in the model. However, some delays can be very system dependent. For example in our case the distance between the reader and the door determines the actual time from a correct entered PIN until the door is opened.

As earlier mentioned certain ranges of values are usually more interesting for the tester. Some values are known before, while others have to be measured or calculated. In our model, we used the log to measure timings in between events and use this data to get realistic timings in the test. The advantages with this model is that will provide very accurate delay from a users perspective, which was one of the goals. But it can also be used as a way to document which timings have been tested and which need more testing. The drawback is that it requires a used system, log-files or data of timings that can be generated from. This is something that only exists later phases of the development of a system in form of system or functional testing.

In the development process, it would be a good idea to start with constant and/or random timings when no data is available to calculate from and once the system is in place, generate delays based on it.

5.4 Automation

Having a high degree of automation for testing will save a lot of resources. A good way to reach that is by adopting MBT. The goal of MBT is to generate test automatically but has

other benefits such as improved traceability, increased readability and understandability of tests etc. Automatic tests tends to have static timings for all kind of systems, even real-time. However, this can be avoided by having a well detailed model to generate your tests from.

A model that contains timings and let each generated test change timings for each testrun is more dynamic. In our method, we have tried to take the automation one step further, by automating the building of the model. Since each PACS will look different depending on its environment, this would be an advantages. The goal is to automatically create a model for any PACS regardless of its environment. For an EFG-based model, this will be the case since it is based on the event-log, and the event-log will look differently depending on the environment. However, the model building is not fully automated. Firstly, a start-state(s) and/or end-state(s) have to be defined manually. By start-state we mean a state that does not have any entering transitions and end-state has no exiting transitions and is considered to be the start or an end of a user case. Without any of these, the graph would most likely be cyclic and we would be able to traverse infinitely long paths. One way to find the start/end automatically would be using a timeout period in other words define the end of an event-sequence if no new event has arrived before timeout is reached. A cyclic graph can still occur if the timeout period is too long. Another approach would be to allow a cyclic graph, but limit the number of times a given event is allowed in a path. However, this would reduce the fuzziness of the generated tests due to limitations of number of stimuli. A third approach would be to simply look for states with no entering or exiting transitions after the model is built and define them as start and end, however one has to be sure that the graph will have states with that property. Since we cannot guarantee that, we chosento with manually pinpoint those states.

Events that contain stimuli made by the Arduino must be manually pinpointed. In PACS the number of events are limited and therefore could be done manually but with a more complex system with a numerous of events, it will be quite demanding. A solution would be to include stimulus information in the event-data from the event-log.

To use a manually created model has its benefits too. The main advantage is that the model can be created without even having a fully working system and therefore used earlier in the development. In comparison to the EFG which requires a fully working and used system for creation of the model. A manual model works good for testing the system's non real-time attributes. It can be used gain a better understand of the system and test it and during the development process the model gets updated to a more sophisticated one.

5.5 Tests

When creating tests for a system, it is often feasible to use a similar level of abstraction for the test as the tested part to get more understandable test. Therefore, in order to replicate a user of the system we decided that all testing will be on an event-based level. Events is the closest type of abstraction we can get to a user. When a user provides some kind of stimulus, one or many events are created. Therefore, the tests will be easy to understand in terms of how the user interacts with the system. However, the drawback of having a higher abstraction is the in ability to test items on a lower level. Lower level errors might still reveal themselves upon testing, but will be hard to track down. It is recommended to

compliment with other testing tools for this purpose.

Having tests based on how user interacts with the system is interesting for many reasons. It can be used for verifying the system for a customer and give the developers confidence about the system's correctness. Quite often when a system gets released to market, the actual usage of the system is not the same as the intended. With usage-based tests, the new non-intended usage would be tested.

Our tests are based on the system's event-log and "mimic" an identical behavior as long it exists in there. This enables the developers to automatically create scenarios in case they want to test something specific or to recreate a discovered error. But it can also be used for exhaustive testing, by creating tests for covering every state in the model.

This type of testing is suitable for any kind of event-triggered system that interacts with an enviroment such as vending machines, ATMs, GUIs etc. The main advantage with the "test suite" is its ability to dynamically generate realistic timings dynamically for every test. By doing this, testers can have few test cases which will be easier to maintain. An important ability for dynamic tests is to be able to reproduce the test in other words store the dynamic part of the test. In our case the JSON-file is stored just in case the test needs to be executed again.

Chapter 6
Conclusion

Today, event-triggered systems is a frequently used architecture and plays an important role in computer systems. For example, Graphical User Interfaces gets controlled by mouse or keyboard events. Other types of event-triggered systems are web applications, video/audio streaming protocols and embedded systems. When it comes to testing, these systems have quite similar properties. Therefore, researches have developed testing methodologies to test them as well. However, for a time-dependent (a real-time system) it is feasible to test time-variations, which these methodologies do well. In this work, we analyzed this problem and solve it by expand an existing graph building technique with timings and probabilistic usage profile. The method is a model-based testing (MBT) and the representation is called Event-flow graph (EFG). An EFG is graph built up according to how the events occur in the system, which can be read from an event-log. A big advantage with MBT is its high degree of automation which makes it easy to carry out automatic tests and which was one of the objectives in this thesis. The "test suite" achieves fully automated test case generation, test execution and test evaluation based on the model. However, the model creation is not fully automatic. For the model creation, a start or end-state has to be chosen. To be able to provide stimuli to the PACS, events involving stimuli must be pointed out manually.

To be able to generate accurate timings, these are included in the model. Each edge has a list with recorded timings from the event-log. From these list we can calculate accurate timings when a test is generated. The list also supplies us with the number of times a certain transitions is taken, which can be used to get a probabilistic usage profile (Markov chain) of the model. If the state space is huge it will be impossible to test every sequence, then the probabilistic usage profile can help us to the most common sequences. Carried out tests can "mimic" a single recorded usage scenario, but also generate approximate scenarios based on earlier usage. Without usages (empty event-logs) the list of timings would be impossible to create and generate good timings. A test is generated by traversing a path from the model. The path will be a sequence of events, and in between each event we generate a timing. We then upload the event sequence together with timings to a stimuli

device that executes the test on the system. The event sequence form the model can be used as the oracle as well. The oracle can automatically decide if a generated test has passed or failed by comparing the traversed path with the latest event sequence in the event-log.

We carried out an evaluation for our implementation on a real system, a Physical Access Control System. In order to carry out this evaluation we used a desktop computer for model building and test generation. As stimulus device, an Arduino was used by sending electronic signals to the PACS. In the evaluation we were able to recreate the exact usage in terms of events and timings, fast and automatic. With well used event-log we were able to recreate approximate scenarios in order to get a higher test coverage.

6.1 Future Work

This thesis suggests several directions for future work. First, it is desirable to have a more effective metric and coverage measurement for a model-based approach. For a MBT based on usage, measurements like code coverages does not give a good value on how well the system is tested, instead requirements coverage would be more interesting. In the future, a more specific metric that could be included in the model would be preferred. Secondly, a more effective model building in terms of automation would be good. Now "important" states such as start/end state and stimulus states must be pinpointed manually, something that would be problematic for a system with a numerous of states. Ideally, the model would be able to discover and pinpoint these automatically. Third, adopt a learning-based approach to the model. In this way, the model keeps itself updated by adding and discarding states or timings depending on changes in the environment. Fourth, study ways to handle cyclic paths in the graph. In our solution we ignore them by not allowing cycles in the graph. Another solution would be to limit the number of times a state can be visited and therefore limit the number of times a cycle is taken. However, in the future, it would be beneficial to have a model that let cycles into its full extend in order to improve the test generation. Fifth, extend the model so it can test the system response times, for example user identification etc. Those timings should be evaluated on other criteria such as MAX/MIN in comparison to stimuli timings. Sixth, by extending the model with application programming interface (API) to do inner stimulus, the testing can get improved even more. By having timings and be able to do both outer and inner stimuli, testers can create advanced tests that would be hard to generate otherwise.

Bibliography

[1] Arduino Ethernet Shield. Webpage. http://arduino.cc/en/Main/ArduinoEthernetShield [11.11.2014].

[2] Arduino Mega 2560. Webpage. http://arduino.cc/en/Main/ArduinoBoardMega2560 [13.10.2014].

[3] BeagleBone. Webpage. http://www.generation5.org/content/2003/FSM_Tutorial.asp [21.10.2014].

[4] BeagleBone. Webpage. http://beagleboard.org/ [18.10.2014].

[5] DSX Access Systems, Inc. What is wiegand? . Webpage. http://www.dsxinc.com/designguide2/docs2/whatiswiegand.pdf [14.10.2014].

[6] JSON: What is it and why use it? Webpage. http://borkweb.com/story/the-case-for-json-what-is-it-and-why-use-it [02.11.2014].

[7] Raspberry Pi wiki at eLinux . Webpage. http://elinux.org/RPi_Low-level_peripherals [13.10.2014].

[8] Techopedia: Functional testing. Webpage. http://www.techopedia.com/definition/19509/functional-testing [30.10.2014].

[9] K. Agrawal and J. A. Whittaker. Finite state model-based testing on a shoestring. *Proc. Pacific Northwest Software Quality Conf.*, pages 154–170, 1993.

[10] Pekka Aho, Nadja Menz, Tomi Räty, and Ina Schieferdecker. Automated java gui modeling for model-based testing purposes. pages 268–273. IEEE Computer Society.

[11] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[12] Mahmoud Awad and DanielA. Menascé. On the predictive properties of performance models derived through input-output relationships. In András Horváth and Katinka

Wolter, editors, *Computer Performance Engineering*, volume 8721 of *Lecture Notes in Computer Science*, pages 89–103. Springer International Publishing, 2014.

[13] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[14] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated analysis of an audio control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52–53:163–181, July-August 2002.

[15] Cambridge University. Reverse debugging study. `http://www.roguewave.com/DesktopModules/Bring2mind/DMX/Download.aspx?entryid=1606&command=core_download&PortalId=0&TabId=607`.

[16] Douglas Hoffman. Getting the most from automated software tests. `http://www.asq-silicon-valley.org/document-for-members/doc_download/303-hoffman-geting-the-most-from-automated-test`.

[17] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison-Wesley Professional, 1st edition, 2009.

[18] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2001.

[19] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *In RT '07: Proceedings of the 2nd international workshop on Random testing*. ACM, 2007.

[20] Wolfgang Grieskamp, Nikolai Tillmann, Colin Campbell, Wolfram Schulte, and Margus Veanes. Action machines - towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*, pages 72–82. IEEE Computer Society, September 2005.

[21] Mats Grindal and Birgitta Lindström. Challenges in testing real-time systems.

[22] Harry Robinson. Model-based testing, google. `https://sites.google.com/site/modelbasedtesting/starwest-2006-mbt-tutorial.pdf?attredirects=0`.

[23] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Formal methods and testing. chapter Testing Real-time Systems Using UPPAAL, pages 77–117. 2008.

[24] Robert M. Hierons. Software testing foundations: A study guide for the certified tester exam. by andreas spillner, tilo linz and hans schaefer. published by dpunkt.verlag, heidelberg, germany, 2006. isbn: 3-89864-363-8, pp 266: Book reviews. *Softw. Test. Verif. Reliab.*, 16(4):289–290, December 2006.

[25] Douglas Hoffman. Using oracles in test automation. In *Proceedings of Pacific Northwest Software Quality Conference*, page 90–117, 2001.

[26] B. F. Jones, H. H. Sthamer, and D. E. Eyres. *Software Engineering Journal*.

[27] Anna-Maija Juuso and Mikko Varpiola. Fuzzing best practices: Combining generation and mutation-based fuzzing.

[28] Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff. Specification-based testing for software product lines. In *SEFM 2008*, pages 149–159. IEEE Computer Society, 2008.

[29] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond, International Workshop, Dagstuhl Castle, Germany, July 8-12, 1991, Proceedings*, pages 87–101, 1991.

[30] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[31] Birgitta Lindström. *Testability of Dynamic Real-Time Systems*. PhD thesis, Linköping University, ESLAB - Embedded Systems Laboratory, The Institute of Technology, 2009.

[32] Karl Meinke and Fei Niu. A learning-based approach to unit testing of numerical software. In *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems*, volume 6435 of *LNCS*, pages 221–235, Berlin, Heidelberg, 2010. Springer.

[33] Gerard Meszaros, Shaun Smith, and Jennitta Andrea. The test automation manifesto. In *XP/Agile Universe'03*, pages 73–81, 2003.

[34] J.R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[35] Brian S. Pak. Hybrid fuzz testing:discovering software bugs via fuzzing and symbolic execution. May 2012. `http://reports-archive.adm.cs.cmu.edu/anon/2012/CMU-CS-12-116.pdf`.

[36] R.S. Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill higher education. McGraw-Hill Higher Education, 2010.

[37] S Prowell. TML: A description language for Markov chain usage models. *Information and Software Technology*, 42(12):835–844, September 2000.

[38] H. Robinson. Experiences in applying statistical testing to a real-time embedded software system. *International Conference on Software Testing Analysis and Review*, 1999.

[39] S Rosaria and H Robinson. Applying models in your testing process. *Information and Software Technology*, 42(12):815 – 824, 2000.

[40] Alex Ruiz and Yvonne Wang Price. Test-driven gui development with testng and abbot. *IEEE Software*, 24(3):51–57, 2007.

[41] Ahmad Saifan and Juergen Dingel. Model-based testing of distributed systems, 2008.

[42] Emil Selinder. System reliability testing of an embedded system. September 2013. `http://sam.cs.lth.se/ExjobGetFile?id=585`.

[43] Ian Sommerville. *Software Engineering 9*. Pearson Education, 2011.

[44] I Tervonen and T. Mustonen. Offshoring test automation: Observations and lessons learned. In *Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on*, pages 226–235, July 2009.

[45] The Institute of Electrical and Eletronics Engineers. Ieee standard glossary of software engineering terminology. IEEE Standard, September 1990.

[46] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid embedded system testing using verification patterns. *IEEE Software*, 22(4):68–75, 2005.

[47] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[48] Antti Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[49] Ilja van Sprundel. Fuzzing: Breaking Software in an Automated fashion. *December 8th*, 2005.

[50] Jeffrey M Voas and Keith W Miller. Software testability: The new verification. *Ieee software*, 12(3):17–28, 1995.

[51] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Softw.*, 17(1):70–79, January 2000.

[52] Xun Yuan and Atif M. Memon. Using gui run-time state as feedback to generate test cases. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

Automatic timing test of physical access control systems

Jonas Klauber
Department of Computer Science, Lund University

December 15, 2014

Introduction

Testing is necessary and vital not only to prove a systems' quality to its user, but it also ensures accuracy, and helps in finding hidden bugs. Over the years, systems have become much more complex, which implies an increased difficulty of creating and carrying out tests. There are plenty of different test methods and approaches, all of which have their pros and cons. However, an important and common method that exists in most of them, is **Automation**. This work suggests a way to generated automatic tests for for a digital system interacting with a non-deterministic environment i.e a human.

Test Automation

Test automation is implemented when standalone programs are produced, in order to execute tests on the system. The programs are capable of conducting tests, reporting outcomes, as well as compare results. Creating automated tools requires a high effort during the building phase. However, once it has been completed, that test can be carried out repeatedly on other programs with minimal effort, at any point in time. In comparison, manually carried out tests require a high effort all the time.

Nevertheless, in spite of its effectiveness, a drawback that accompanies automation is that such tests usually run with identical configurations and timings each time. In a system with user interaction, the inputs and timings of a test will vary. In such cases, automatic tests would require a test case for every combination inputs and timings, which could be very expensive, if not an impossible task. With that in mind, even if automatic testing is well established in the industry today, it is still uncommon to conduct automatic tests for user behavior. Currently, this type of testing is performed almost entirely manually, and it can be very time-consuming and expensive. At the same time, efforts to increase test coverage by defining new tests are usually ignored due to time constraints. By creating a model that can generate dynamic tests, significant time and resources can be saved.

Our System

In this thesis, we look at a physical access control system (PACS), where just like any access system, a user is able to swipe his card, enter the PIN-code, access a door. A non-deterministic part (a user), provides an input. The input and timings differ between users; i.e a user may swipe a correct or incorrect card, or answers a phone call while holding up the door. In order to mimic the users' behavior, a microcontroller (Arduino) is used to send electrical signals to the PACS to simulate different inputs and outputs (e.g. swipe card, enter PIN-code, etc.).

Lastly, an automation tool is needed to produce a list of inputs and timings that describes a desired user behavior. The Arduino then executes the list and reports the outcome for the test.

Test Generation

To produce a list of inputs and timings, we need to design a way to describe typical usage of the system. One way would be to draw a sketch of the system usage, which is the most common approach. For an access system, this would be quite straight forward. However, there would still be the issue of accurate timings.

A better approach would be to "mirror" users' behavior, where it would enable us to create the exact input with the real timings. As most systems use logs mainly for debugging purposes and safety reasons, so does the PACS. In the system logs, the system usage is described in detail. From these logs, a graph is created that will describe a typical usage of the system, such as the order of inputs, and the delay between them. With this information, we can build a more accurate abstraction of the system that can be used to generate dynamic tests. The result of the dynamic tests can then be compared and evaluated against the graph.

Conclusions

During the development of a system with a non-deterministic third party, which is often the case in a event-based real-time system, there are many advantages of using such a test system. It can create tests that copy an earlier usage of the system. Each test run will be different in order to increase the test coverage. The graph also gives a useful overview of the events and the order they are created.

There are two drawbacks with this approach. One, logs need to be available before the test can be created. Secondly, it is not completely automatic, and there is still a need for supply of data, such as starting nodes and user data.

Overall, a testing approach like this has a lot of potential and is a great complement for testing especially for control and quality verification purposes. Although this may be a simple test of "mirroring" a third party, by extending the possibility to do external changes in the systems configuration, developers can create test cases that otherwise would be hard to cover.