Faculté des Sciences
Département d'Informatique

UNIVERSITE LIBRE DE BRUXELLES,
UNIVERSITE D'EUROPE

Practical case of Black-box driven verification

Thomas VAN GYSEGEM

Directeur : Gilles GEERAERTS

Mémoire présenté en vue de l'obtention

Du diplôme du Master en Sciences informatiques

Année académique 2016 - 2017

abstract

This paper will describe a practical case of software testing and explain all the choices that where made during the implementation process. It will also try to provide some generic way of doing the same for a different use case given a similar running environment and constraints.

Remarque générale : ajouter des phrases (intro / conclusion) de "liant" pour donner une idée du contexte général.

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence

---

Edsger W. Dijkstra , The Humble Programmer (1972)

Contents

1 Introduction

There are a lot of available softwares nowadays but have you ever wondered where they all come from ? All of them come from some development process. Those processes can last for days, months or even years before producing the expected software and during this process, some errors can be made that makes the software faulty or broken. That's why, all along this process, we need to make sure that everything is working just fine. To do that, we introduce a step in the development process called "testing". Without it, no complicated software will be able to do its task without problems. Of course, this testing step can be done differently depending on the overall development process but nevertheless, every software need to goes through that step.

Not so surprisingly, this paper is all about this particular testing step. The goal of it will be to provide a way of automating this step. As we will explain later on, providing this automatic process is of major interest in term of time gained in the development process and reliability for the final software. Of course, detecting all problems during tests is a really complex task and the process developed throughout this paper will only help to reduce the risk of shipping faulty softwares and to make sure that previously encountered/corrected problems are not re-introduced between different versions of it.

To guide us during this work, we will focus on a concrete case study that was encountered at the Railnova company in Brussels. Of course, as the goal of this work is to stay generic, we will go from the concrete examples solution to more generic solutions that can easily be extended for any other software.

Despite the fact that this paper will try yo stay generic, we will not talk about hardware testing even if we can see that as a particular case of testing in a development process. Despite the fact that strong timing constraint doesn't allows us to performs this hardware analysis, we think that focusing on software development will help the reader to get a better understanding of the principles used and developed ideas.

I tried to divided this work in many big parts that introduces them-self one after the other. As this topic and all the techniques involved are closely related, there will often be references to area not already covered during the

sequential reading of this paper.

The first part of this work will describe the existing techniques in software testing and discuss of what are some of the motivations of this work to introduce Finite State Machine (FSM) and how they work. Following this, we will discuss about the work that was done, why it is an important improvement and what are the decisions that lead us to this particular implementation. Then, we will explain how we designed the testing process and finally, we will describe how our particular case was handled by explaining how we integrated every inputs and outputs of our software into our testing process. We'll also give a more summarized/generic way of doing the same work for any kind of inputs and outputs.

Part I
Preliminaries

To begin with, we will present the very basics of software testing, what are the common practices and how we use them nowadays. This will allows us to introduce the technical terms used all along this paper. Following this technical introduction, we will present the studied case that was the starting point of this work at the Railnova company to introduce FSM. Finally, we will describe what is a FSM exactly and some of its variations.

2 Software development and software testing

Software development is a complex area and a lot of different processes exists but at the very last, they all have some key principles in common. One of them being the testing part. For this work to be fully understood, we need to introduce a bit of software development and then, explain which are the different techniques used in software testing. This will provide us the required technical vocabulary to describe our own testing process.

Software development can be defined in a lot of different ways, every one of them being more or less equivalent but in this work, we will focus on the basis and simply say that software development is some kind of plan to produce a given software. This plan can be more or less complex and must involves at some point, some testing steps.

Software are usually ordered by a client that can be ourself, some company or any set of users [1]. Clients have needs and our task as software developer will be to fulfill them by producing a software. The client's will first be translated into requirements specifications that, to put it simply, are formal description of them. We usually create those requirements specifications during meetings with the client, this requires a lot of talk to be sure that the idea we get from them is what they want.

Then the software can be built. Of course, if we got the wrong requirements specifications, our software will not be what the client asked and therefore, they will want us to start again from scratch. Lets assume that those requirements are good, then, once the software is built, we want to test it to ensure that it works and that it meets the given requirements. Once those test are successfully passed, we can ship the software to our client.

This whole process, starting from the client order to the shipping of the produced software is what we call the software development process. Of course, there is not one only way of doing it even for the most simple process like this one their can be divergence of work-flow from one team to another. Lets introduce some of them [2].

2.1 Waterfall

The waterfall model is probably the most basic development process still used today, it is often attributed to Winston Walker Royce in a paper from 1970 [3]. The whole process is sequential or plan driven. The whole process is divided in big parts and each of them is done only one time and one after the other. The original model was designed as follows:

- System and software requirements: captured in a product requirements document
- Analysis: resulting in models, schema, and business rules
- Design: resulting in the software architecture
- Coding: the development, proving, and integration of software
- Testing: the systematic discovery and debugging of defects
- Operations: the installation, migration, support, and maintenance of complete systems

The following problems can easilly be spotted in this process:

- If an error is made during the analysis step, this error is repercuted to the next steps.
- Between the first and the last step of it, there can be a lot of weeks, months or years ! During this time, the project needs can evolve and then, the delivered product will only match outdated requirements, which, of course, is not a good thing for the client.
- If some problems occurs during the writing of the product requirements, let's say that the client forget to talk about a key feature. Then, again, the product we will deliver will miss that feature.

Producing the wrong product is really a big problem of this methodology and because of all those problems, organizations/teams tends to avoid it when possible. Nevertheless, there is still a lot of qualities to this process. As the first steps are strongly related to documentation of the product, if a team

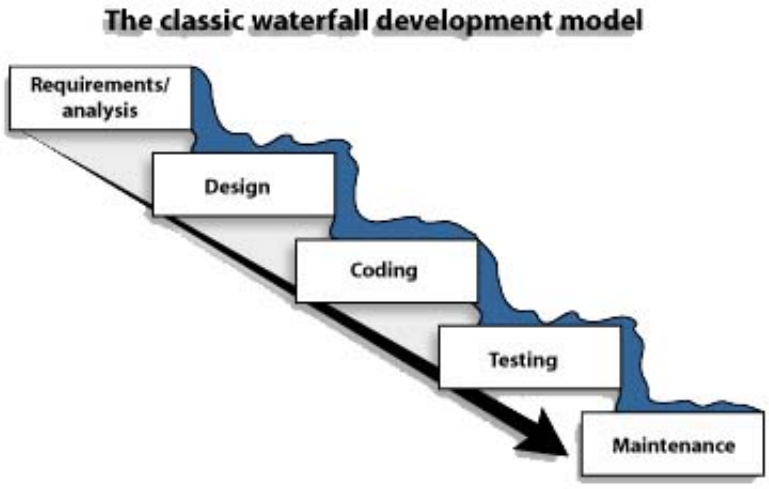

Figure 1: Waterfall model. You should see why it is called waterfall from this picture.

member leaves during the process, we have less risk of loosing knowledge of what has been done or what is left to do. New team members will simply need to read the requirements, the produced models and the software architecture.

This process is also one of the easiest to understand, every step of the project is well documented and one can easily jump in any project using this methodology and be ready to participate in a very small amount of time.

2.2 Prototyping

Sometimes, you want to produce a product as soon as possible so you can conquer the market sooner. In that case, the waterfall approach is not going to be very useful. The prototyping process though, can help you towards this goal.

The idea is to produce some prototype as soon as possible, test it and then, iterate over it again and again until the final product is fully functional. It is a trial-and-error process. This works especially well when the final product requirements are not fully known. By presenting a prototype to

the client, he is able to give feedbacks and explain the following most wanted features. Like for the waterfall model, we have a plan for the whole process: The whole product is divided into as many sub-product as needed. Each of those division will be prototyped on top of the last ones until we get the final product.

- System requirements are defined in as much details as possible.
- A first design is created.
- The first prototype is created.
- Client tests/reviews the prototype.
- Developers change the prototype accordingly.
- We repeat those steps as many times as needed.
- The final product is constructed based on the last prototype.

Using this model, the probability to deliver a product that does not meet the client requirements is very small as he can give a feedback throughout the whole development process. But there are some drawbacks to that.

Focusing on small sub-problems/sub-features distracts the developers from conducting a good and in depth analysis. This is also due to the lack of fully specified requirements from the start. Adding more and more features on top of existing ones can quickly create a complicated product both difficult to maintain and improve.

2.3 Agile

Agile Software Development is based on twelve principles:

- Customer satisfaction by early and continuous delivery of valuable software
- Welcome changing requirements, even in late development

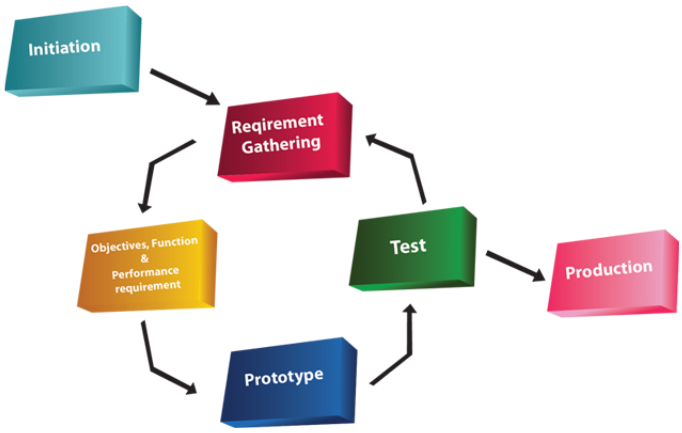

Figure 2: Schema of the prototyping process.

- Working software is delivered frequently (weeks rather than months)
- Close, daily cooperation between business people and developers
- Projects are built around motivated individuals, who should be trusted
- Face-to-face conversation is the best form of communication (co-location)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Continuous attention to technical excellence and good design
- Simplicitythe art of maximizing the amount of work not doneis essential
- Best architectures, requirements, and designs emerge from self-organizing teams
- The team reflects on how to become more effective, and adjusts accordingly

There exist many different frameworks that complies to those principles. One of the most commonly used is Scrum [4]. With this framework, we

work using sprints. A sprint is the basic unit of development in Scrum. The team fixes the time-frame during which the sprint must be done and then, fixes the scope of work that needs to be done during a Sprint Planning event.

At the end of each day, the team holds a Daily Scrum to answer the following questions:

- What did i do ?
- What will i do to meet the sprint goals ?
- Is there anything that block the team progress ?

Finally, at the end of the sprint period, the team holds a Sprint Retrospective to answer the following questions:

- What went well ?
- What could be improved ?

This allows for a very dynamic development process in which no ones get stuck too long on ineffective or impossible processes.

2.4 Validation and verification

Now that we have settled down our requirements specifications, we need to introduces the validation and verification topic. Validation and verification is what we use to make sure that our software is meeting the previously specified requirements. In short, that our software is exactly what we specified. Failing to this would produce a software that will maybe not meet the clients requirements and, therefore, be of little or no use to him.

Both concepts are closely related but are completely independent Software validation is a procedure used to ensure that the software meets the user's needs and that the specifications were correct while software verification is a procedure used to ensure that the software follows the specified requirements [5]. Following is a more formal definition:

- **Validation** is the confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled [6]

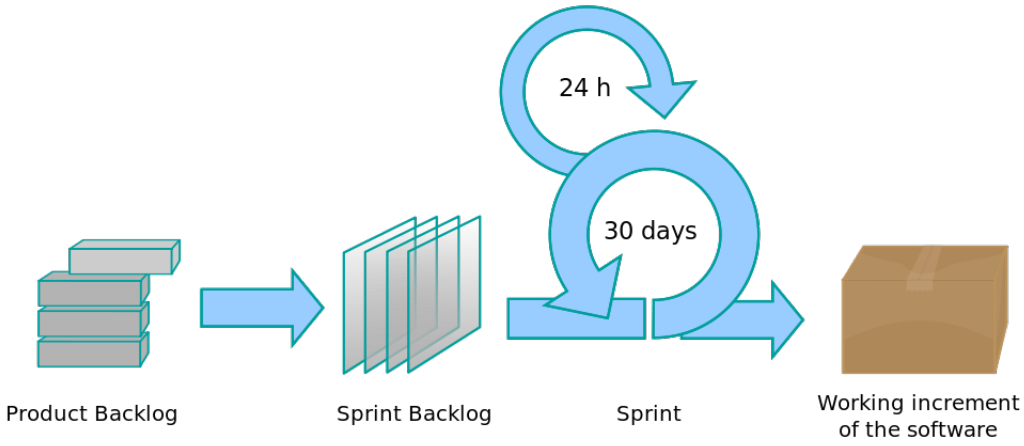

Figure 3: Schema of the Scrum Framework One of the most effective development process. This is only an example, the sprint duration is not always 30 days.

- **Verification** is the confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled [6]

Simply put, validation is answering to "Are we trying to build the right thing ?" while verification is answering to "Have we made what we were trying to make ?". Of course, in our case, we are more interested in the verification part and we will leave the validation part out of this work. We will assume that the requirements are correct and therefore that if we follow them, the built software will be the one our client wants.

2.5 Code coverage

Code coverage is a metrics used in software testing that basically tells you how much of your code has been tested. It can be computed on many different criteria like how many subroutines have been called, how many lines of code have been interpreted and so on.

The idea behind this metric usage is that, if all of your code is tested, then there is probably no bugs left, or at least, there is a good probability

that we will encounter fewer bugs in a completely tested software than in an untested one. Getting a good code coverage is therefore a key point in testing as it improve the software quality by reducing the probability of faults left in it [7].

2.6 Black-box and white-box testing

Black-box testing and white-box are two terms used by softwares testers to describe how they approach their testing [8]. They describes two of the most common used techniques in software testing.

With black-box testing, also known as functional testing, we consider the software as a black-box and we have no knowledge of what is inside of it nor on how it works. This is the opposite of white-box testing where the whole software is known from the inside, all the code is accessible and we have a deep knowledge of how it works.

Both of those approaches have their own advantages and drawbacks. In white-box testing, the tester is testing functionalities that are already developed and he can miss needed features that have been forgotten. He has a very good code coverage as he can produce test cases for every specific details by reading the code. White-box provides an internal perspective of the software.

Black-box testing, on the other hand, provides another point-of-view and does not require a deep knowledge of the code to operate. Test cases are designed following the requirements and not the code itself so if all the tests are successful and that all the requirements are entirely covered by the designed test, we have a proof that our software meets the requirements.

White-box testing is more or less done by developers themselves while black-box testing can be performed by anyone with poor programing skills. Code coverage can not be used while doing black-box testing for the obvious reason that you don't know what is done internally so you don't know what part of the software code is interpreted.

2.6.1 Grey-box testing

There is also a third option there which is called "Grey-Box testing". We are doing Grey-Box testing when we do both Black-Box testing and White-Box

testing.

2.7 Static and Dynamic testing

Two other terms used in software testing are static and dynamic testing. Static testing is the process of reviewing something that is not running. For example, some compilers (like gcc) perform static testing by finding unaffected variables, unused variables, dead code, syntax errors and so on. Dynamic testing, on the other hand, is performed while the program is running.

2.8 Type of testing

There exists a lot of different types of software testing, explaining them all is way out of the boundaries of this work so we have chosen 6 of them that are closely related and can be spanned on a whole software development process [9].

2.8.1 Unit testing

Scope: White-box testing
Who: Coder

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. The goal is to isolate each part of the program and show that the individual parts are correct [10].

In a more practical point-of-view, it's about isolating each function/method-/class/... from the code and test their behavior for a wide range of input values. A lot of faults can stay undetected after this step. One of the reason of this being that people who write unit-test have a deep knowledge of the underlying code. This knowledge influence them when writing unit-test so it becomes really hard to set realistic test cases. Also, if those who write the unit-test are the same ones that wrote the code that is being tested, the test code is likely to be as faulty as the code being tested itself !

Listing 1: Brief example of what unit testing is (C pseudocode)

```
/**
 * @brief Reads a file content
 * @param name Name of the file
 * @param buffer Buffer for the file content
 * @return 0 and buffer is filled with the file content.
 *         -1 if file does not exists
 *         -2 if file can't be read
 */
int readFile(const char* name, const char* buffer);

// Tests fake file name
assert(readFile("doesnotexist.txt", buffer) == -1);

// Tests file can't be read
assert(readFile("busyFile.txt", buffer) == -1);

// Tests file read success
writeFile("file.txt", "some text");
assert(readFile("file.txt", buffer) == 0);
assert(buffer == "some text");
```

2.8.2 Integration testing

Scope: Black-box and white-box testing
Who: Coder

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing [11].

2.8.3 Functional testing

Scope: Black-box testing
Who: Independent tester

Functional testing is about testing each functionality of the whole system as described in the requirements.

Functional testing tests a slice of functionality of the whole system. Functional testing does not imply that you are testing a function/method of your module or class. If the functionalities wanted are:

1. Sending invoices to clients
2. Creating invoices
3. Adding clients

Then you will have test cases that try to add clients, another one that try to send invoices and acknowledges that the client, preferably a fake one, received it and a last test case that will create various different invoices.

2.8.4 System testing

Scope: Black-box testing
Who: Independent tester

System testing involves putting the new program in many different environments to ensure the program works in typical customer environments

with various versions and types of operating systems and/or applications [9]. System testing is testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements [6].

This is very relevant in computer science where every computer is built from different component and can have many different operating systems. To perform system testing, you will usually run the program on different operating systems (Unix based systems or DOS based systems) and also on different hardware configuration. Of course, testing every possible combination is not possible so some exotic systems can still prove themselves unable to run the program after this steps. Of course, if your program is built to run on a specific pre-defined system, this testing will be really straightforward.

A basic example of system testing would be two computer, one running a Unix based operating system and the other running Windows for example (any version will do). In this case, the system test would be to run the program in both environment (Unix based OS and Windows) and check that, given the same inputs, both of them produces the same results or at least, similar enough ones. In other words, verify that they both works as expected.

2.8.5 Acceptance testing

Scope: Black-box testing
Who: Client

Acceptance testing is formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system. More formally, the system is tested to verify that it meets the given specifications or contract.

Usually, the user will provide some test or use case for each desired functionality. You can add your own test to those but at least the client test must be satisfied before trying to deploy the product otherwise the client will probably not even try the product.

2.8.6 Regression testing

Scope: Black-box and white-box testing
Who: Coder

Regression testing is a type of software testing used to ensure that a software that was previously developed and tested still meets the previously defined requirements after a modification has been introduced into it such as a bug fixes or enhancements. The purpose is to make sure that the newly introduced modifications do not damage any previously-working functionality.

Practically speaking, we will only add new test when we are adding new functionalities and we will remove test only when the functionality covered is no longer needed or maintained. Then, if all the tests are still running without error on our new software, we will assume that previously developed functionality still works.

2.8.7 Beta testing

Scope: Black-box testing
Who: Client

This step is performed when the software is fully or well developed. The software is shipped to a limited amount of users so that they can test it as they wish. The purpose is to detect fault by providing inputs that testers have not thought about because of their advanced knowledge of the software and its purpose.

2.9 Requirements-based testing

Requirements-based testing is a testing approach in which test cases, conditions and data are derived from requirements [12]. Those test can be performed from the start of the development process and to the end of it [13]. Of course, if the requirements are not good, those test will be of little use. Good requirements are critical but requirements checking is, again, out of our scope. So we will simply assume that we have good requirements for this work.

Requirements-based testing is exactly what we are trying to do with black-box testing. For each given requirements stating, for example, that we want to produce some result Y when you give some input X, we want to produce a test case that maps the input X to our software and gets the output Y'. Comparing Y and Y' will allow us to decide whether or not, there is a fault and/or decide if the requirement is met or not. If not, our software is not built correctly and we need to work more on it before shipping it for the client.

All the previously described type of testing are not necessarily requirements-based. For example, unit testing is about the code. We want to make sure that each part of the code work well but satisfying that does not imply that requirements are met ! For a 3D modeling software, we can have a code that compute square roots, if the code works well, that does not imply that our software allows the user to design houses for example.

3 Formal Verification and Model Checking

Our goal is to be able to verify that any kind of product, software or hardware, is working as expected. Formal verification can help us to achieve this goal. It is a process that allows us to check whether or not a given model satisfies given properties. You can, for example, create a model of a given software and verify that it does not contains bug. To be able to do that, both the model and the specification must be formally described in a mathematical language.

The model is a simplification of the real system which can have some properties. The requierements we want to check must be expressed as properties which can then be checked against the model using FSM that we will now introduce.

3.1 Finite State Machine

A FSM is a mathematical model of computation. It is a mathematical structure used to provide an abstract description of the behavior of a system. This mathematical structure is composed of a list of state. A FSM can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs. The function that maps state and inputs to states is called a transition. A FSM is defined by its list of states, its initial state, the input alphabet and a set of transitions [14].

The input alphabet is the list of every symbol it can receive in any given state. For example, an input alphabet could be a list of keyboard's key if we consider a FSM that simulates the behavior of a text editor. The key points here are that you need to have a finite set of state and a finite set of transitions.

Definition 3.1. Finite alphabet. A finite alphabet Σ is a finite set of symbols.

Definition 3.2. State-transition function. A state-transition function δ is a mapping $\delta : S \times \Sigma \to S$ that maps each state s with a symbol a from an

input alphabet Σ and another state s'. [1]

Definition 3.3. Finite State Machine. A FSM is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the finite input alphabet.
- S is the set of state.
- $s_0 \in S$ is the initial state.
- δ is the state-transition function.
- $F \subseteq S$ is the set of final states.

Because of those rules, a transition function can start and end in the same state, thus creating a loop on the state. δ is conventionally allowed to be a partial function. This means that $\delta(s, e)$ does not needs to be defined for every possible combination of $s \in S$ and $e \in \Sigma$. This allows us to have a finite δ transition function even when the input alphabet Σ is not finite or defined.[2]

About the input alphabet, we usually define every input as a character or token read on the input tape of the FSM. Where the input tape is a sequential list of every symbols feed into the system. This is a lot like a Turing Machine.

Definition 3.4. Finite word. A finite word $\sigma = \sigma_1\sigma_2...\sigma_n$ is a finite sequence of symbols from Σ: $\forall i : \sigma_i \in \Sigma$

Definition 3.5. Transition table. A transition table $\mathcal{A}$ is a tuple (Σ, S, s_0, Δ), where:

- Σ is the input alphabet.
- S is the set of state.
- $s_0 \in S$ is the initial state.
- $\Delta \subseteq S \times S \times \Sigma$ is the set of state-transitions function.

[1] This hold only for deterministic FSM, we'll see later how to express δ for non-deterministic FSM and what non-deterministic/deterministic FSM means.

[2] This is crucial for our practical case describe later on.

Definition 3.6. Run. A run r of the transition table $\mathcal{A} = (\Sigma, S, s_0, \Delta)$ over a finite word $\sigma = \sigma_1\sigma_2...\sigma_n$ over the alphabet Σ is the sequence of state

$$r : s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} ... \xrightarrow{\sigma_n} s_{n+1}$$

where $\forall i \geq 1 : (s_{i-1}, s_i, \sigma_i) \in \Delta$

Let us extend the definition of FSM with another usefull notion:

Definition 3.7. (Extended) Finite State Machine. A FSM is a tuple $(\Sigma, S, s_0, \delta, W, L)$, where:

- Σ is the finite input alphabet.
- S is the set of state.
- $s_0 \in S$ is the initial state.
- δ is the state-transition function.
- $W \subseteq S$ is the set of final accepting states.
- $L \subseteq S/W$ is the set of final failure states.

This extension of finite state machine doesn't introduce any more complexity. Take the original definition where a finite state machine FSM is a tuple $(\Sigma, S, s_0, \delta, F)$. In this definition, the set of final state F is equivalent to $W \cup L$ from our extended definition.

With this new defintion in mind, we define a run to be an *accepting run* and *failed run*: Every run is a said to fail if it is not an accepting run. To determine what is an accepting run, assume at least one state s_i from r is in W: Take the one with the smallest i. If i equals 0, the run is accepting, if not: if $\forall j < i : s_j \notin L$ then its an accepting run.

3.1.1 Description of Finite State Machine using graphs

Having a more human friendly way of presenting FSM is possible using graphs. To show an FSM as a graph, we create a directed empty graph G. Then, we create a node for every state of our FSM. We need to remember the mapping node/state. Various solutions can be used for this problem: Let us apply a function that give each state label starting at 0 and incrementing by one for each state. Then, lets label every newly created node with the

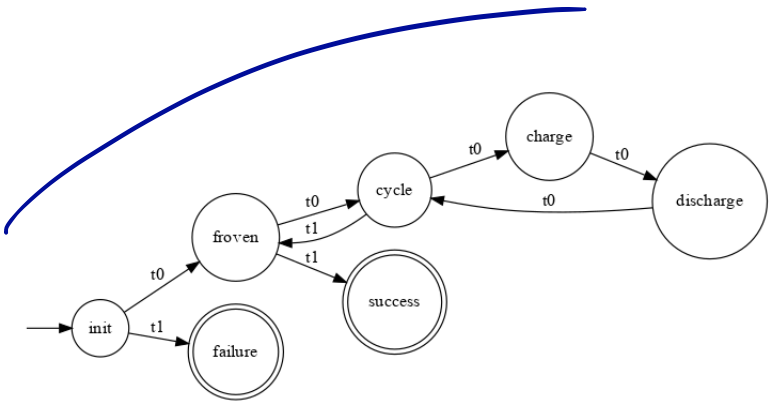

Figure 4: Example of a FSM shown as a graph.

label previously assigned to the state.

Now we add an arrow for every transition defined in δ with the starting point being the current node and the endpoint is the node corresponding to the goal of the transition. Finally, we add an arrow starting from nowhere and pointing to the node representing the initial state.

Every state s in W or L is displayed with a double circle. We label those in W with "*success*" and those in L with "*failure*". This gives us a graph we can display like in the figure 4. For comprehension sake, we have also labeled each node with a brief state name and every transition with an unique name based on the starting node.

3.2 Acceptors

An acceptor takes a sequence of inputs from the input alphabet and run a FSM on it. Every state of the acceptor is either and accept state or a reject state. When the computation ends in an accepting state, we say that the input is accepted and the input is not accepted otherwise (figure 5). The set of sequence of input symbols accepted by an acceptor is called a regular language.

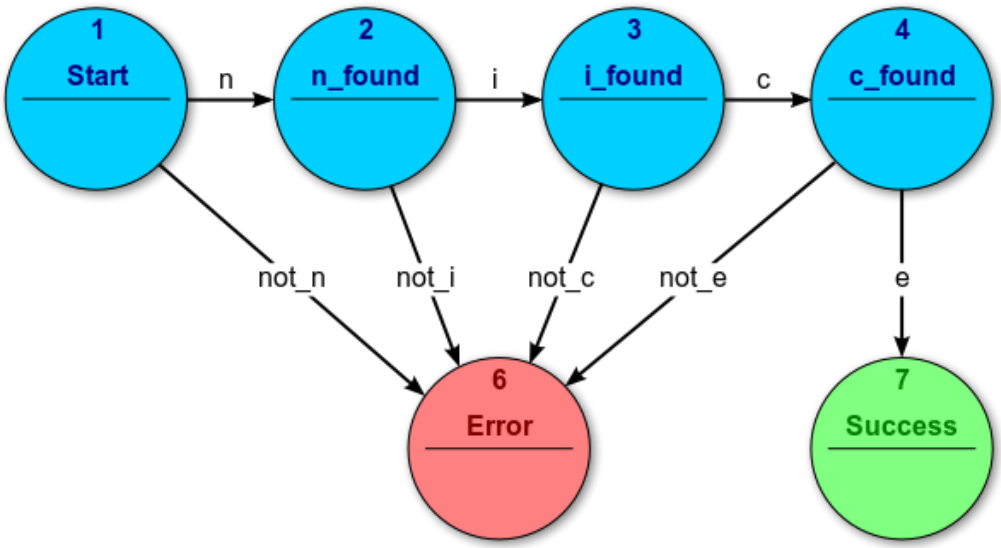

Figure 5: Example of acceptor for the string 'nice' [14]

3.3 Non-Determinism

A Non-Deterministic Finite Automaton (NFA) is a kind of FSM in which transition does not requires the reading of an input symbols. We can take transitions from one state to another without looking at the input tape. The decision of taking such a transition is done in a non-deterministic way. And some transition can also lead to more than one state. So reading a given symbol x in state $q1$ can lead to $q2$ or $q3$ for example (see figure 6). [15]

Formally speaking, we represent a NFA with a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$ where:

- Q is the finite set of states
- Σ is the input alphabet
- Δ is the transition-function $\Delta : Q \times \Sigma \rightarrow P(Q)$ where $P(Q)$ is the power-set of Q
- q_0 is the initial state with $q_0 \in Q$
- F is the set of accepting states where $F \subseteq Q$

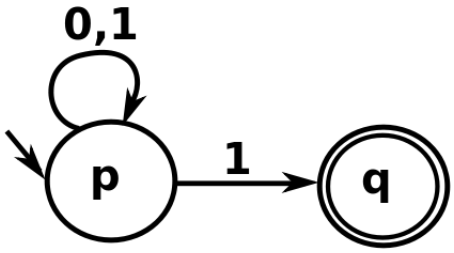

Figure 6: Example of a NFA. In state p, reading the symbol 1 can lead to p or q [16]

In an NFA, the input alphabet Σ usually contains a special symbol for representing a transition that can be taken without reading the next input symbol on the tape. This symbol is ϵ.

It is well known that a Deterministic Finite Automaton (DFA) is equivalent to a NFA. So using one or the other will not alter our ability to solve given problems.

3.4 Alternating Finite Automaton

An Alternating Finite Automaton (AFA) is a 6-tuple, $(S(\exists), S(\forall), \Sigma, \delta, P_0, F)$, where:

- $S(\exists)$ is a finite set of existential states
- $S(\forall)$ is a finite set of universal states
- Σ is a finite set of input symbols
- δ is a state-transition function $\delta : (S(\exists) \cup S(\forall)) \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^{S(\exists) \cup S(\forall)}$
- P_0 is the initial state with $P_0 \in (S(\exists) \cup S(\forall))$
- F is the set of final state with $F \subseteq (S(\exists) \cup S(\forall))$

In an AFA, transition are more complex than those we encountered in traditional FSM. The transition we are working with can lead to more than one state ! This is used to represent logical AND and OR with transitions

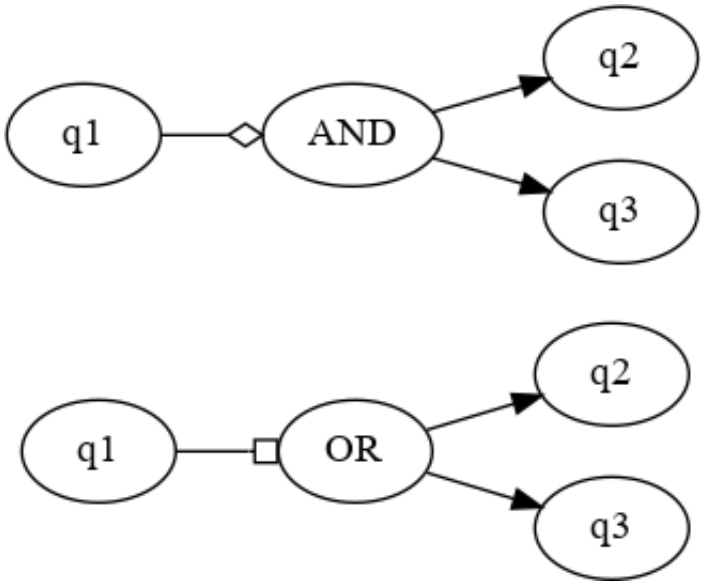

Figure 7: Example of a AFA.

(see figure 7).

To implement this automaton, we need to be able to store the system's current state and restore it later so we can do multiple run and back-track when needed for every existential and universal transition we have. When we take an AND transition, we need to run a second automaton in parallel for each state the transition leads to and when we encounter an OR transition, we behave like a non-deterministic automaton by taking one of the possibilities non-deterministically. Depending on the number of those transitions, this increase the memory usage of our automaton as well as the time needed to complete the evalutation of all possible paths.

The computation of such an automaton is a tree. And deciding whether or not an AFA accepts a word w if there exists a run tree on w such that every path ends in an accepting state. [17]

3.5 Regular Language

We define a word over a given alphabet as a sequence of symbols from that alphabet. As you already know, a FSM takes an input alphabet as a parameter and every sequence of symbols from that alphabet are words. We also define a language as a set of words. A FSM can accept or reject words, we will call the set of accepted words a regular language over the input alphabet of the FSM.

More formally, we define the language recognized by our extendred finite state machine:

Definition 3.8. Language. The language $L \subseteq \Sigma*$ recognized by an automaton is the set of all words whose run over the automaton are accepting.

3.6 Timed Automata

To define a timed automata, we need some more definitions. A timed automata works with timed sequences, timed words and timed languages.

For a given input alphabet Σ, a timed language is defined as a set of timed words over Σ. A timed word over Σ is a pair (σ, τ) where σ is a word over Σ and τ is a timed sequence. Lastly, a timed sequence is a sequence of real time values increasing monotonically. [18]

Definition 3.9. Timed sequence. A timed sequence is an inifinite list of values increasing monotonically $\tau = \tau_1\tau_2...$ where $\tau \in \mathbb{R}$ and $\tau_i > 0$ and $\forall i \in \mathbb{R} : \tau_i < \tau_{i+1}$.

Definition 3.10. Timed word. A timed word over an alphabet Σ is a pair (σ, τ) where $\sigma = \sigma_1\sigma 2...$ is an infinite word over Σ and τ is a timed sequence.

Definition 3.11. Timed language. A timed language over an alphabet Σ is set of timed words over Σ.

We also need a transition table that can read timed words. We want the transition to consider the input symbol from the input alphabet and the time associated to it when it makes a transition choice. To do so, we associate a set of clocks to our transition table and for each transition, we associate a clock constraint to it and require that the transition may be taken only if the current values of the clocks satisfy this constraint. A clock can be set to zero simultaenously with any transition.

& examples

Definition 3.12. Clock constraint. A clock constraint is defined inductively by

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

with $x \in X$ and $c \in \mathbb{Q}$. X being a set of cloks.

Definition 3.13. Clock interpretation. A clock interpretation v is a mapping $X \to \mathbb{R}$, with X being a set of clocks.

A clock interpretation v over X satisfies a clock constraint δ over X iff δ evaluates to true using the values given by v.

Definition 3.14. Timed transition table. A timed transition table $\mathcal{A}$ is a quintuple (Σ, S, s_0, C, E), where:

- Σ is the input alphabet.
- S is the set of state.
- $s_0 \in S$ is the initial state.
- C is a finite set of clocks
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ gives the set of transitions. A transition from the state s to s' on input symbol a is represented by a qintuple $(s, s', a, \lambda, \delta)$ where λ is the set of clock to be set to zero and δ is the clock constraint over C.

We now have all the needed tools to define properly the type of timed automaton we'll need for this paper:

Definition 3.15. Timed automaton. A timed automaton (TA) is a tuple $(\mathcal{A}, W, L)$ where $\mathcal{A}$ is a timed transition table, $W \subseteq S(\mathcal{A})$ (with $S(\mathcal{A})$ being the set of state of $\mathcal{A}$) is the set of *success* states (W stands for winning as S was already taken) and $F \subseteq S(\mathcal{A})$ is the set of *failure* states.

Definition 3.16. Run. We define a run r of a timed automaton TA $(\mathcal{A}, W, L)$, where $\mathcal{A}$ is (Σ, S, s_0, C, E), over a timed word (σ, τ) as the sequence

$$(\overline{s}, \overline{v}) : (s_0, v_0) \xrightarrow[\tau_1]{\sigma_1} (s_1, v_1) \xrightarrow[\tau_2]{\sigma_2} (s_2, v_2) \xrightarrow[\tau_3]{\sigma_3} \ldots$$

with $s_i \in S$ and v_i is a mapping $C \to \mathbb{R}$ $\forall i \geq 0$.

- s_o is the initial state of $\mathcal{A}$.
- $\forall x \in C : v_0(x) = 0$
- $\forall i \geq 1 : \exists e \in E$ of the form $(s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i)$ so that $(v_{i-1} + \tau_i - \tau_{i-1})$ satisfies δ_i and v_i equals $[\lambda_i \mapsto 0](v_{i-1} + \tau_i - \tau_{i-1})$. [18]

Intuitively, a run is the sequence of states and clock interpretations characterizing the *run*/processing of a given timed word to a given timed automata.

Example 3.1. Timed automaton. An example of timed automaton where $C = \{x\}$ is given in figure 13.

We can now speak of successful and failing runs. Given a timed automaton TA $(\mathcal{A}, W, L)$ and a run r, lets assume r contains one or more (s_i, v_i) where $s_i \in W$. Take such (s_i, v_i) with the smallest possible value of i. The run is said to be successful iff $i = 0$ or $\forall j < i : s_j \notin L$. We define all the other runs as failing runs.

Using this new definition of run, we can define the language accepted by a timed automaton TA as the set of timed words whose run over TA are accepting:

Definition 3.17. Timed language. Given a timed automaton TA $(\mathcal{A}, W, L)$, where $\mathcal{A}$ is (Σ, S, s_0, C, E), the language $L \subseteq \Sigma*$ recognized by TA is the set of timed words which produces accepting runs.

3.7 Applications

One of the first question one should ask is how is model checking and formal verification used in the everyday life or at least, for some projects. The following examples of what is currently used in this domain where taken from the most used tools at the time this paper was written. Unfortunately, this choice is rather subjective as metrics about each possible software usage is not always available but the only thing that interest us is to know how they works and what they have in common.

3.7.1 UPPAL

The first software we came across while searching for such applications was UPPAL. UPPAAL is a rather complete tool to work on model checking and

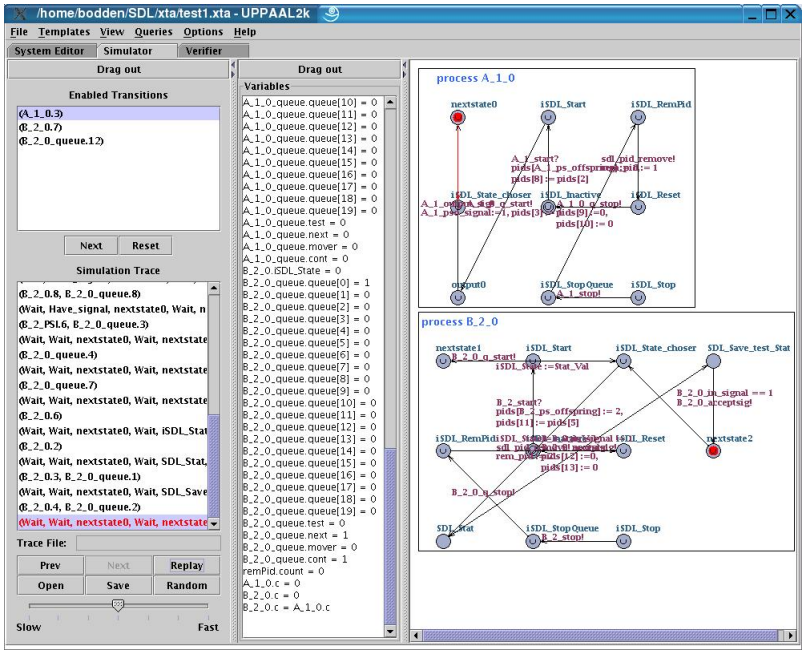

Figure 8: Main window of UPPAAL for the simulation part. You can see the execution trace in the lower left corner of the window as well as the model of the system on the right, shown as a TA

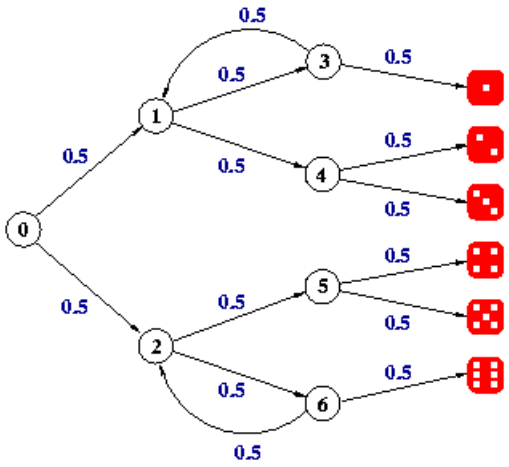

Figure 9: Example of model used by PRISM. This model was generated using the lines of code in figure 3.7.2.

formal verification. With it, one can modelize a system, simulate and verify it. All the three steps in only one tool [19] [20] [21].

UPPAAL was a great source of inspiration for some features of our own tool. We were specifically interested by the simulation tools which allows to navigate through the execution steps and visualizing what is going on with the model of our system. Also, UPPAAL uses TAs which comforted us in our choice of such tool.

3.7.2 PRISM

We where also interested by PRISM, which is a tool for modelization and verification of probabilistic systems. PRISM supports different kind of probabilistic models like Discrete-Time Markov chain (DTMC), Markov decision process (MDP), Probabilistic Automaton (PA), continuous-time Markov Chain (CTMC) and Probabilistic Timed Automaton (PTA).[3] [22] [23]

[3]Probabilistic Automaton are Markov Chains that also supports deterministic transitions.

Listing 2: "Dice model in PRISM"

```
dtmc

module die

// local state
s : [0..7] init 0;
// value of the die
d : [0..6] init 0;

[] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
[] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
[] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
[] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
[] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
[] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
[] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
[] s=7 -> (s'=7);

endmodule
```

PRISM uses a uniform modeling language for all the probabilistic models that it supports. This is a textual language, based on guarded command notation. See figure 9, this model was generated using the following lines of code:

Again, this tools uses FSM to modelize different systems.

& Conclusion : explique comment ces models vont être exploités.

Part II
Case study

Now that you know what software testing and FSM are all about, lets introduce the case that is of interest to us, namely, Nursery ! I'll first introduce the Railnova enterprise and the system we will work with all along this paper. Then I will explain why this work is of critical importance for them and generally speaking, for every software development process.

4 Railnova

This work is done in partnership with Railnova. Railnova is a technology company that provides telematics and software solutions to the railway industry. Their headquarter is located in Brussels and they count approximatively 30 employees. Some of them coming from the ULB/VUB.

The main goal of this work is to develop a better testing suite for their embedded software called RailsterOS. Of course, the developed solution will not be limited to a particular software and our scope is a lot wider than that. Our solution will be considered successful if it meets the following requirements:

- The system can be used by non-developers
- It is a time saving technology
- The solution reduces costs

4.1 Existing system

RailsterOS is a software developed at Railnova which purpose is to collect and stream data from a train to a server. This data is then processed to give various insights about the train status. Thus allowing Railnova's client to manage their fleet with ease.

The RailsterOS is executed on a dedicated hardware also built by Railnova which is called Railster Universal Gateway (RUG). Those RUG are mainly installed in trains in Germany, France, Belgium, Netherlands and Spain at the time this work was written. More specifically, the RUG is installed in the engine of trains.

This dedicated hardware provides a lot of data from the train. We can access informations like remaining fuel (if it is not an electrical engine), the total running time since last power on of the engine, where is the engine located, current speed if the engine is moving and all the data that the client wants to access remotely.

Every engine is more or less unique. They can use a wide range of different component and thus have different parameters like their maximal speed when loaded or not, the maximal weight they can carry or even the number of wheels the engine uses. Because of that RailsterOS needs to accept a wide range of different configuration so we don't stream fuel remaining if we are on an electrical engine for example. We can also have two kinds of motors that give information about their speed without using the same type of cable or protocol. To achieve this flexibility, we allow RailsterOS to start only a subset of its internal software using some configuration files. All this software's modules, which we call daemons, are small applications used to interface with a given protocol. For example, we have one daemon that is dedicated to GPS location's informations. The set of daemon we use, which is defined in the configuration files, defines what type of engine we can handle and what operations we can do.

To retrieve all those informations, we need to use different communication protocols and wires because engines built by different companies will have different behaviors and communication protocols. All of them, we will call interfaces. Each of those interface needs proper test that relies on the used protocol, we want to be able to describe our needs by saying things like "I want this particular data when I receive this kind of message on this interface". As there is a lot of them, this will take time to implement proper testing for all of them so we will only cover a subset of them in this paper but we will provide a generic way of implementing test for them all.

4.2 Railster Universal Gateway

As stated before, RailsterOS is running on RUG. Those hardware are built by Railnova. Actually, the majority of them are in their first version (RUG1) but the RUG2 is on its way. It is important to keep that in mind to understand why we tried to keep this work as generic as possible. Indeed, in a year, maybe less, the majority of the product used will be RUG2 so if the system can only work for a dedicated and specific hardware, we will need to start over all this work for the RUG2.

RailsterOS is being developed incrementally since the start of the company. It means that the software is kind of messy with all the new features coming in and the old ones being deprecated or deleted. Actually, there is

more than 10 different versions of the software in production and those are not the only existing versions.

It is frequent to encounter new bug after the release of a new version that were supposed to be fixed since earlier versions simply because the new changes re-introduced the problem. This is mainly because Railnova is a young company, a start-up, and the work priority are very end-user scheduled. If a system stops working, a quick fix is developed and pushed to production as fast as possible to avoid loosing too much money but the cost on the long run for this operation can be greater than what we saved by fixing the problem quickly.

Say that the problem is fixed by a quick and dirty patch. Then, one year later maybe, a major problem arises because all the fixes deployed are preventing us from adding wanted features. Then, the whole system is to be re-developed from the start. That was what happened when they tried to put RailsterOS on the RUG2. RUG1 was developed the quick and dirty way to be on the market as soon as possible. So a lot of conception errors were made. The software had been modified to a point where it needed all those conception errors from the RUG1 to operate correctly.

4.3 Short story

Some of the RUG were experiencing troubles during the end of fall 2016. 8% of the fleet had to be replaced by the team for unknown reasons. After a bit of testing, some members of the team thought that it could come from an unexpected electrical over-consumption of the RUG while connecting to a GPS antenna. To validate this hypothesis, some testing had to be done. It was the end of summer so the outside temperature was going down so the team thought that maybe it had something to do with that.

The test procedure was defined as follow: First, set the temperature of a thermionic chamber to -10 Celisus to replicate the outside temperature recorded when we experienced the failures. Put a RUG in it and connect to the serial communication port of the system. Once we are connected, we issue some command to manually choose which GPS protocol to use out of the four of them supported and for each of them, we manually set the emitting power of the GPRS chip in order to reproduce the problem seen on the

practical field.

Those test were made, approximately 30 for different temperatures of the thermionic chamber, different band and different emitting power. Those took one person an entire day of work ! That's because the thermionic chamber needs an hour to set a fixed temperature, the system needs 5 minutes to reboot when we set the band to use and we wait 10 minutes for each test to be sure the problem did not appear (or did). After each test, if we want to do another one, we need to wait for the system to cool down as each time we boot it up, it produces heat.

And those tests are just to prove the initial hypothesis. Once this was proved, we wanted to know for sure, what emitting power we could use for each RUG (not only for one) depending on the band used and the external temperature. This would need hundreds of tests if not a thousand on a lot more than just one RUG to be able to make enough data and say for sure which levels are safe, which levels are risky and which levels are forbidden to avoid problems in the real world.

And that's without mentioning that the thermionic chamber makes a lot of noise to work, wasting an entire room as people can not work long in such a noisy environment.

This is where the Nursery project comes in handy. The goal of it is to provide an environment fully automatized to realize all those testing when we want, like during the night so the room stays silent during the day for men to work in, without needing one or more persons to take weeks or years of works.

5 The economics of software testing

Usually, when you work on a development project, you have many different development stages. The main ones are the proper development of the product and the final stage of the product, when it is shipped to the production environment.

Shipping the product to production simply means that you put the prod-

uct into its real environment. In other words, you begin to manipulates real world data and not the tests sets that you where probably using during the development. In our case, we install a RUG running RailsterOS in some engine.

But if the shipped product contains faults, it will probably begins to produce incorrect results and, depending on the product, it can even harm people or destroy things by using them incorrectly. The client will not be able to use the product he paid for to its full capabilities and will maybe want to switch to another company that produce more reliable products. Its easy to imagine other cases in which having fault in the final product must imperatively be avoided.

All of this comes with a cost, for example, we can imagine that we are working on a assembly line controller. If our controller (product) does not work correctly, the assembly line can be damaged because of incorrect usage or the materials being manipulated can be damaged too resulting in repair cost for the assembly line and incorrect/damaged product assembled that cannot be sold afterwards.

In the RUG ecosystem, if our product, RailsterOS isn't working well, the RUG its running on will not send the intended data to the client. For him, it is like the engine itself is malfunctioning or not functioning at all if we are unable to send any data ! He will then need to send a team to the engine to find the problems and a lot of work will be delayed because the engine is not running and, of course, every step in this process is very costly for the client and for the company itself because when a product in production is faulty, we also need to get back the devices that are malfunctioning and, perhaps, every other device running the same software of having the same hardware. And finally, we need to replace them and invest time in correcting them or, at least, solving the issue for the other product versions.

It is commonly believed that the earliest a fault is found, the cheaper it is to correct it [24] [25]. The case we mentioned is the latest possible time to detect a fault so, according to the previous statement, it is also the most expensive ! Those reasons are why we want to create some process that can be used to drastically reduces the amount of unexpected faults in the final product. That's the main motivation behind the project that is the subject

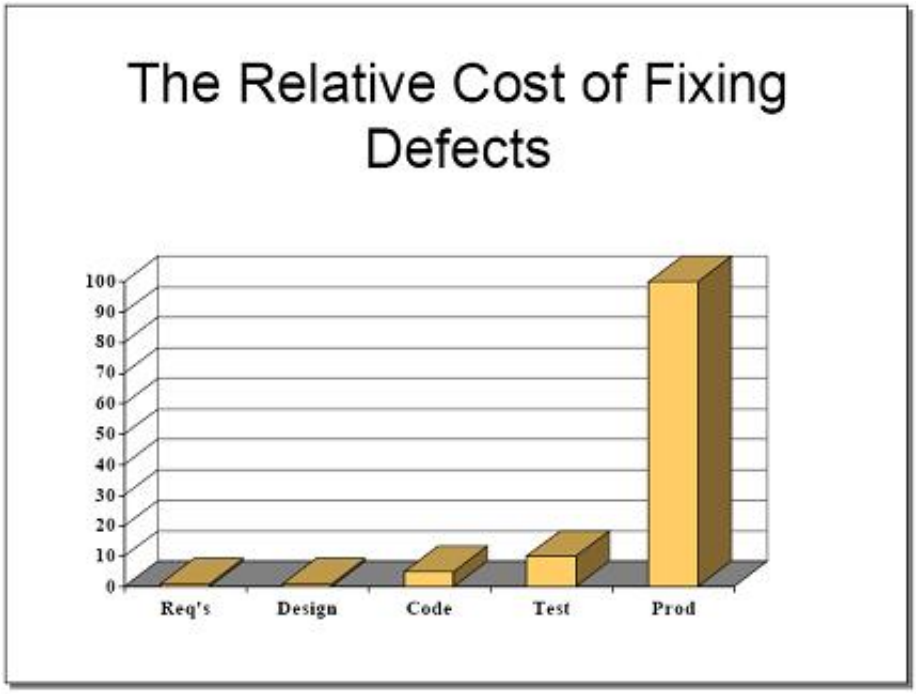

Figure 10: Relative costs of fixing faults [24]

of this paper called Nursery.

5.1 The Railnova Way

At Railnova, development in general is lead by the Agile framework. Because of that, softwares and hardwares developed tend to welcome changes and new features are added every week as well as small bug fixes or minor modifications there and there.

This tend to make it difficult to keep track of what is working and what is not during the life-cycle of any project. This is where Nursery becomes useful. By providing a tool to check whether or not features are broken after modifications have been applied would be a great time saver. Testing everything manually is a tedious work and waiting for things to break in production environment is a very bad idea like we saw previously.

Usually, when one of the RUG fails in production, the product is called back to the Railnova HQ for Fleet Quality Management (FQM). FQM consists of several phases. During the first one also called "fingerprinting", we try to identify the root cause of the failure. Then, we re-direct the product either to "Debug HW" for hardware debugging if the root cause is hardware linked or to "Debug SW" if the root cause is software linked. After those steps, the problem should be solved. The product is sent to "Fixing" to apply other pending modifications that did not necessitate an immediate return of the product. Finally, the product is sent to "Flash & Validate" to re-initialize it completely and flag it as "ready to ship".

Part III
Contribution

All the problems described above provides us a lot of informations. We know what can go wrong, we know what we need and we know that a proper validation of all the developed product is a key for success of any given product on the long run. So having a global view of what software testing is and what is the concrete problem we are dealing with, we will try to design a testing process that can be applied to it and possibly to any other given problem that needs black-box testing. One of our main goal is also to design a tool that will be useful and easy to use but we want it to be powerful enough to be able to handle every test case we can think of.

6 Requirements analysis

First of all, lets take back our software development concepts introduce during the first chapter of this paper. We have described seven type of testing and now that we know what we are working on, lets try to summarize what type of testing we really need:

- Unit and integration testing: Those kind of testing are great to test the code but that does not give us much informations in terms of requirements. Knowing that every functions works separately is not enough to tell that the requierements are met as the program behavior results from the interaction of all those function one with another.
- System testing: We do want to make sure that our interfaces can run on different hardware. In our case, we want to be able to validate our software for at least RUG1 and RUG2.
- Functional testing: Testing that the software fullfils the requirements is, of course, our main preocupation. So this type of testing is of main importance.
- Regression testing: Being able to certify that a new feature or a bugfix does not break the whole system is a key point too. We dont want to re-introduce old bugs or break what is already working well.
- Acceptance testing: Automatizing client testing is not really in the scope of this paper. Besides, we assumed earlier that the requirements are good and well validated.

Summed up, those are the 3 main requierements we absolutely want in our solution:

- Functional testing
- System testing
- Regression testing

We now want to know what we need to satisfy those requierements. But first, let us introduce some concept that will be used to formalize our following ideas:

Definition 6.1. Our software/Our program. We will talk a lot about "our software" or "our program". All those notation refers to the software being tested.

In our case, our software refers to railster-os which is the Unix based operating system we want to test.

Definition 6.2. Requierements. Requierements R is a set of requierement defined by our client(s). Keep in mind that we can be our own client.

Definition 6.3. Testing. A test t is used to map a particular requierement $r \in R$ to a satisfaction value:

$$T : r \to \{True, False\}$$

When mapped to $True$, we say that the requierement is fullfilled and the requierement is not fullfilled when mapped to $False$.

This definition is still incomplete, we need to know how the mapping is done and that is the question we are going to answer in this chapter.

7 Functional testing

Given a set of requierements, we want to make sure every one of them is fullfilled, using the definition we stated earlier, this is pretty simple to formalize:

Definition 7.1. Functional testing. Functional testing is used to ensure that the following condition holds:

$$\forall r \in R : t(r) = True$$

8 System testing

Now, you know what system testing is about. We want to verify if our program can run on more than one system. At Railnova, the practical case we had at hand was about two different hardware, RUG1 and RUG2, both of them handles inputs and outputs differently and we wanted to make sure that our custom Unix based OS would run as expected on both of them.

Of course, the newest version, RUG2 had more features than the RUG1 and this lead us to our first observation: As different systems don't have the same capabilities and given that all capabilities should be tested, see functional testing if you don't recall why, the set of test we should run on a given system is system dependant.

Again, let's take a practical example: The RUG2 can handle more communication protocol than the RUG1 and can process more informations in the same amount of time. So our custom OS should be able to handle those new informations when run on the RUG2 but this is not requiered when run on the RUG1. To fullfil this need, we now know that our solution will have to provide a way of defining the subset of all existing requierement to be tested on a particular systems.

Definition 8.1. Systems. $Systems$ is the set of all possible environment on which we want our program to run.

Definition 8.2. System. A system $System \in Systems$ a particular environment on which we want our program to behave as expected

Definition 8.3. System requierements. The subset of requierement that are required for a given $System$ is $R(System)$

Definition 8.4. Testing. We can now extend our definition of test by including the system on which the test is done:

$$T : r, system \rightarrow \{True, False\}$$

Definition 8.5. System testing So now, system testing is simply making sure that, given any $System \in Systems$, the following condition holds:

$$\forall r \in R(System) : t(r, system) = True$$

9 Regression testing

In regression testing, we want to make sure that he previous testing are, in the worst case, as efficient as before. If we only maanged to fulfill a subset of the needed requierements, we want that this subset of requierement is still fullfilled in the newest revision of our software.

Definition 9.1. Revision. We are going to iterate on the tested software to improve it gradually. A revision V of the software s is a mapping of the software to :

$$V : s \to \mathbb{R}$$

with the following conditions:

- s is the latest software revision
- s_0 is the first software revision and $V(s_0) = 0$
- $\forall x \in \mathbb{R} : V(s_x) > V(s_{x-1})$

Using that, lets extend our test definition to include the software revision:

Definition 9.2. Testing. Given a requirement r, a system s and a software s_x:

$$T : r, system, V(s_x) \to \{True, False\}$$

Having defined those notations, we now have:

Definition 9.3. Regression testing Verify that, given any $System \in Systems$ and a given software revision x, the following condition holds:

$$\forall r \in R(System) : t(r, system, V(s_x)) = t(r, system, V(s_{x-1}))$$

Basically, what worked before must still work now.
// TODO INPUTS OUTPUTS
////
One of the problem we are focusing on is what do we need to build a reliable process that can reduce drastically the odds of shipping incorrect or faulty products to our clients ?

To reply to this question, we need to analyze what we need or what are our requirements. We can begin by saying that we need to handle a set of inputs and a set of outputs. We also need rules to parse the set of outputs and validate them or not.

What we mean by saying "validating the outputs" is simply determining if those are the expected outputs or the correct outputs. If not, we encountered some invalid outputs meaning that our product is faulty and cannot be

shipped.

For the RUG, we have a really huge amount of different inputs sources and all of them can produce a wide range of various outputs. So, for this work to be done in time, we choose to only cover a given subset of those inputs/outputs. But, we cover them, we'll try to give some general ways of handling them so this work can easily be extended to all kind of inputs/outputs regardless of the project being considered.

10 Testing environment

The testing environment should emulate perfectly any given train/engine. Each train have a set of interfaces that are plugged into the tested system. What we mean by "interface" is one particular input type, those will be described later on. The system that will emulate the train (or the testing environment) will be given a set of "scenario"

Those scenario will describe what can happen to the system and how it will react to it. A strong constraint in the case that concerns us is that, those scenario will be written by people with poor or no knowledge of advanced programming so we need to provide a simple yet powerful language to write those stories.

We will assume that our testing environment have a way of sending every kind of inputs to the software being tested and retrieved every kind of outputs from it. Of course, in some case this is very difficult to achieve. Lets consider a practical example that is happening for the RUG case.

The RailsterOS communicates with a GPS/GSM chipset that is connected to antenna that are not under our control. This is an issue because under those circumstances, if the antenna can behave differently in production than during our tests sessions so some faults can goes undiscovered until that.

Some solutions exists for this kind of problems like having our own antenna or using development boards for this chipset. We could also try to write some kind of simulator that will interface with the GPS/GSM chipset but

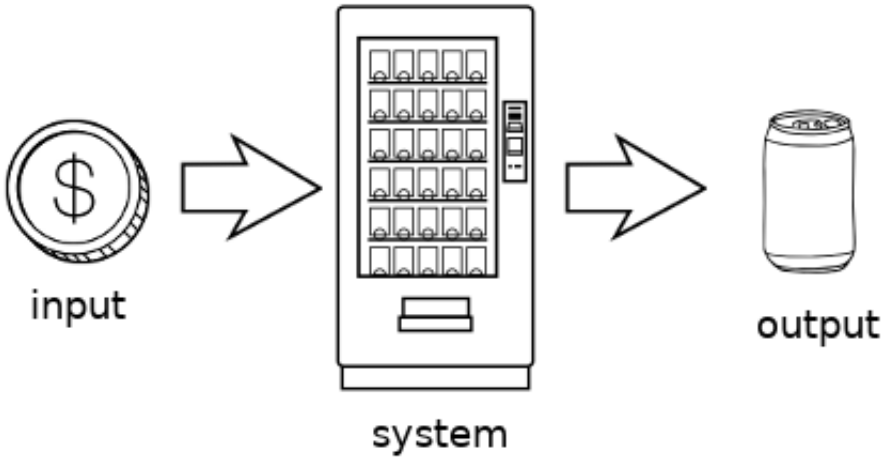

Figure 11: The vending machine I/O mapping

we would need to reverse engineer the whole GSM/GPS networks or at least one particular antenna which would be really time and resources consuming. Not to mention that it will also probably be way more expensive than fixing hypothetical faults when they arise in the production environment. So the complex interfaces needed for those kind of technical component and specifically, their implementations are not a simple matter and are way beyond the scope of this paper so we will not talk more about them.

In this paper and for case like this one, we will simply consider that it is not a part of the system that we can test. So every component that are part of the whole system being tested are provided with some interface that allows us to communicate with them, sends and retrieves data.

With that in mind, we want to be able to describe what happens on the input side of the studied system and then, what are the expected results or outputs produced. For example, when you put some money in a vending machine to buy a soda, the system is the vending machine, you don't know how it works (or at least, most people don't), you feed it some inputs (some money) and you expect a given output (a soda) (figure 11). This is exactly what you want to describe in your tests.

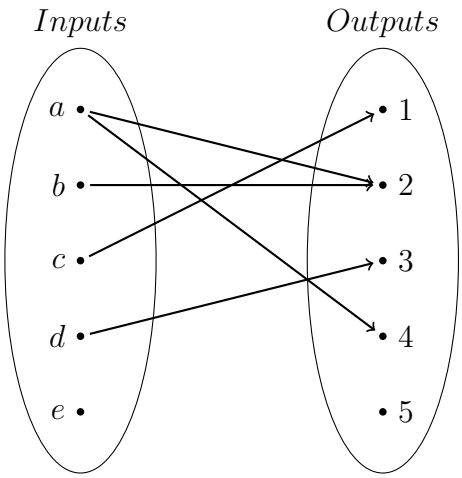

Figure 12: Not explicit schematisation of a valid mapping of inputs to outputs, all possible cases are not taken into account. On the left, the input we send into the system and on the right, the outputs we can receive. Arrows shows what are the expected outputs for a particular input

A naive idea to do so could be to write a file mapping every possible inputs to every possible outputs and claim that our process works only when no unpredicted input/output pair occurs during some run of it (figure 12). But then, what if we want to produce a different output depending on how many times we seen some given output ? With this first idea, it would not be possible to handle this as we don't keep track the current state of the process. We don't have a way of counting how many given inputs have been encountered since the start of the process.

This first idea gave us an important lesson, we need to keep track of the state of the process or at least of what happened before the current time. This does correspond to what we call FSM and this is why we introduced them earlier.

11 Choices

When asked what is the type of FSM that fits our needs the best, we want to give the best possible answer. To do that, we need to rank the different possibilities that we have introduced based on the following criteria. First, we want to make sure that a particular FSM type is possible to apply in our case. If it is an abstract FSM that uses infinite memory or another paradigms that is impossible to implement on modern computer, we know that we can discard it.

Then, how easy it is to describe a system's behavior with such FSM ? If it takes to much times or if we find it too difficult for non-technical peoples to describe a system's behavior using that particular FSM type, we know that it will probably be deprecated over time and wont be used as it should be.

Lastly, if the chosen FSM introduces restriction on what can be described, those restriction must not prevent us from doing every testing needed. And if it introduces new possibilities that allows us to describe more system's behaviors, the complexity that it introduces into the system should be worth it.

Those three criteria are the ones that where chosen as the most relevant for the choice of the FSM that will be used.

- Is it possible to implement ?
- Is it user friendly ?
- Does this introduces restrictions ?

Lets review the possible choices we have and justify the one we picked. But first, let's explain what are the main differences between our practical case and theory. In theoretical FSM, the number of state is upper bound by the system's internal memory. If we want to test every possible state it would gives us 2^x states to analyze (where x is the system's internal memory expressed in bit). In modern computers, this memory can be around two to height gigabits maybe and this is raising every passing years. So having to describe what should happen in $2^2 * 10^9$ different states is not really a possible choice.

To address this problem, we need to abstract the state definition. We'll aggregate formal states into a more complex structure that we will still call a state but for the user, it will just be something like "initialization state", "error state", "processing data state" and so on. A state will simply be a set of computation that needs to be applied. Three optional function, one for the computation needed when we reach the state, one for when we leave the state and the last one for the computation we need to do every time a new symbol is read.

The input alphabet Σ formally defined earlier is also problematic, how can you describe every input received by a modern computer ? That requires very low-level knowledge and a lot of work. We want the user to be able to describe every states and transition without minding the Σ input alphabet. To achieve this, lets define Σ as a set of abstract inputs instead of character's set or token's set.

Now to the choice of FSM model, why don't we simply pick the most simple case of FSM ? The reasons are very simple, take this case: You want to be able to detect a failure that happen if you receive a given data more than 10 seconds after some given data. How can you do this ? Or even simpler, you want to detect a failure if you didn't receive any data for 30 seconds.

All of those case requires you to create a new input type that will be the time. Say you receive this input each time a second pass. Already, it is a lot like timed state machine but lets say we want to use a pure FSM. Now, we need to add state for each seconds passed, so if we want to wait 30 seconds, we need 30 states with 30 transitions for each seconds and 30 transitions for the event of "a data has been received" that will go back to initial state. And all of this needs to be written by someone. You can easily see why this is impracticable. So clearly, in this case, FSM are too limited if we want to handle this in a simple and practicable way.

Now with the AFA, the complexity of implementation required by this paradigm is quite important, we need to run multiple automaton, construct execution trees and analyze them. Compared to a simple FSM implementation, this is quite a lot of work. So is it worth it ? To answer this question, we need to find practical case where such a paradigm would become indispensable and such case should be really difficult to implement into a classic

FSM to justify choosing AFA as our main choice for the implementation of Nursery. Based on the concrete needs we add when we analyzed this possibility, we choose not to consider it as a good solution because none of the case we add prepared needed such complexity and using this would have made it difficult to design test by inexperienced users.

This left us with timed automaton. This is the choice we made. Those FSM are really well adapted for our use case. Being able to manipulate time directly in our transitions and state gives us a lot of ease to write the code and its also very powerful too ! With it, we are able, for example, to send a lot of data with a variable interval between them and then receive data accordingly. You can also decide to create a clock for timeout that will be reseted each time you receive a given data. There are really a lot of use case and some more will be described later on.

This choice of tool is also motivated by the fact that writing and understanding automaton is a very easy process. Designing a User Interface (UI) to create automata and generating valid scenario out of it is a relatively easy process so this could be a way to meet the strong requirement that our solution shall be easy to use for not seasoned programmers.

As an example, take the pseudo-code showing how test are designed using this tool in listing 3. We create 4 states, one, "init", is purely optional, it's purpose is for initializing external tools or some variables if we need them. Then a state that will simply do nothing, waiting for some data to arrive. A success and a failure state. This test works by looping in wait-state every time a data is received. The test success when enough time has elapsed without any timeout and it fails at the first timeout encountered.

As we modified a lot the FSM principles, we'll say that our FSM succeed or fails when the test passed or failed. We will not say that our test accepts nor recognizes as we think this is not applicable for our case.

12 F

ormal definition

Listing 3: "Pseudo test case example"

```
class MyTest:
    add_state("init", enter=init_all)
    add_state("wait")
    add_state("success")
    add_state("failure")

    init.add_transition("wait", True)
    wait.add_transition("success", no_failure_since_long)
    wait.add_transition("failure", timeout)

    set_initial_state("init")

    listen_to("distant_data")
```

Theres a lot of modifications we did to the basic concept of FSM. First of all, our input alphabet is never quantified. We don't know if it's finite or not. Theoretically, we accept analogical inputs so it can be infinite but as we are working with computer, we know it will not be infinite. The most important thing is that we don't need to quantify it to design our test, the choices we made keeps us from being forced to enumerate one and every possible input. Like we said, it will define itself based on what transition are defined in δ and a special symbol for a time step, let's say t.

Also concerning the states, they are not simple mapping of the system's internal memory state. They are more complex abstraction of an arbitrary user defined state. And finally, our δ transition function is allowed to be partially defined, every undefined transition will have no consequences on the FSM computation.

12.1 Determinism

Explain: Transition not on symbols but on complex conditions First, evaluate complex conditions (mapping to TRUE/FALSE) Maps each transition to a transition number $\mathbb{R}$ unique/growing. Find first transition number mapped to TRUE -¿ transition taken Explain why it's deterministic

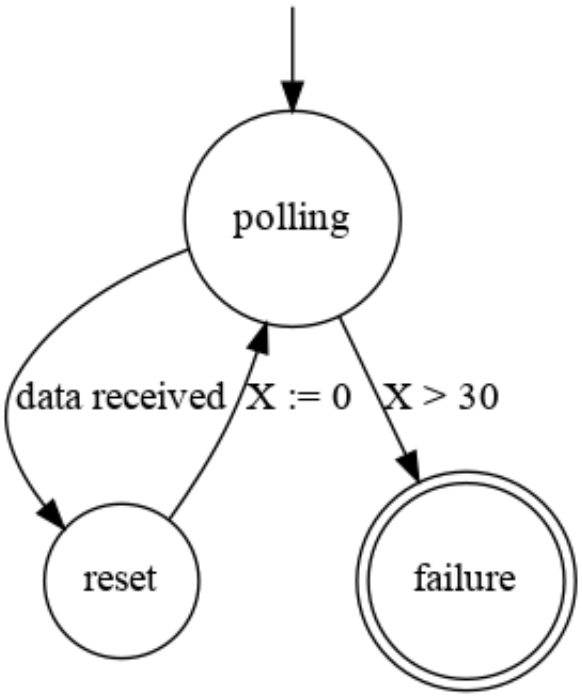

Figure 13: Example of a timed-out timed automata. X being one of the clocks that can be used

Part IV
Results

This part is a in depth description of how the implemented system works, how it has been developed and what it can and cannot do. This is a more technical part dedicated to the implementation where we will not talk a lot about the mathematical concept behind it as those have already been explained before.

13 Nursery

Because of the platform imposed for this project, which is Raspberry Pi, we had the choice between C or Python for the implementation. We choose Python as it is easier to deploy, easier to write and easier to debug. Of course, the produced software is a bit slower than if it where written in C but this is not a limiting factor in our case.

13.1 Architecture

Nursery works on top of two main features, logging and reporting. Logging allows us to keep track of Nursery outputs. For every test, a file is created to log every one of them. Each line of output have an additional property which is the level of urgency of the output. The possible levels are the following:

- DEBUG. Used to output values that are not vital to understand what is going on but are of absolute necessity for debugging purposes.
- INFO. The default output level. Simple information messages.
- WARNING. Used for warnings. When an expected error occurs which is not broking the system.
- ERROR. Error messages. For the messages that should not happen.
- CRITICAL. When the system is broken, this kind of message should be used.

The goal of this logging feature is to produce files that contains every single informations needed to identify the reasons of test failure and contains all the tests parameters. A lot of time was put into this feature to allow for useful context related informations for every line of output which, we find, is a key component for Nursery.

On top of this logging facility, we added a reporting feature. This one is intended to update bug trackers, product tracking facilities or anything else needing informations about the status of a particular hardware or software. Typically, this is used to update a product tracking server, Redmine, used at Railnova to keep an history of what happened in fingerprinting. When a test is done, the results is added on Redmine by saying: "Test ¡name¿

success/failure on RUG ¡serial number¿", of course, other informations are added such as the test version, test criteria and why the test failed. This can also be used to keep track of the status of a production line. At every stage of the production line, we can update the status of the product being build with this reporting tool.

Each type of reporting is described by a Python file. Adding one is a very easy process that any developer can achieve. To match the correct reporting to the test being performed, we specify a product type. Those are pre-defined in a configuration file that will be described later on. Each product type maps to a particular reporting type. For example, if we say that we want to perform a test on a product of type "RailsterOS", on the software then, we can have a reporting that will send reports by mails or send them on a bug tracker.

13.2 Configuration

The whole systems is based on one configuration file. This file contains all available test for Nursery. It also contains all output providers, all batch of tests and the list of all product that can be tested matched with their corresponding reporting. An output provider is a script that everyone can write used to provide data to the test. We used it, for example, to gather data in an online database. Those data are then send to each test as we receive them. A batch of test is just a list of test that needs to be performed in the given order.

Like we say earlier, we also have a concept of product type. Each possible product type is mapped to a given reporting file using a code similar to the one shown in the listing 7. The list of "tests batches" are listed using the code of listing 6 and finally, each test is defined using a code similar to the listing 5.

Several parameters are needed to add a test. **name** and **description** are simply used to display documentation about them. **l_product** is a list of product to which the test applied. This is purely used to warn the user if he tries to perform a test meant for RailsterOS on a battery pack for example. The duration specify the maximum duration of the test in second. After that, the test fails automatically. Finally, the **l_scenario** parameter is the most important one. It specifies all the FSM that are run in parallel to

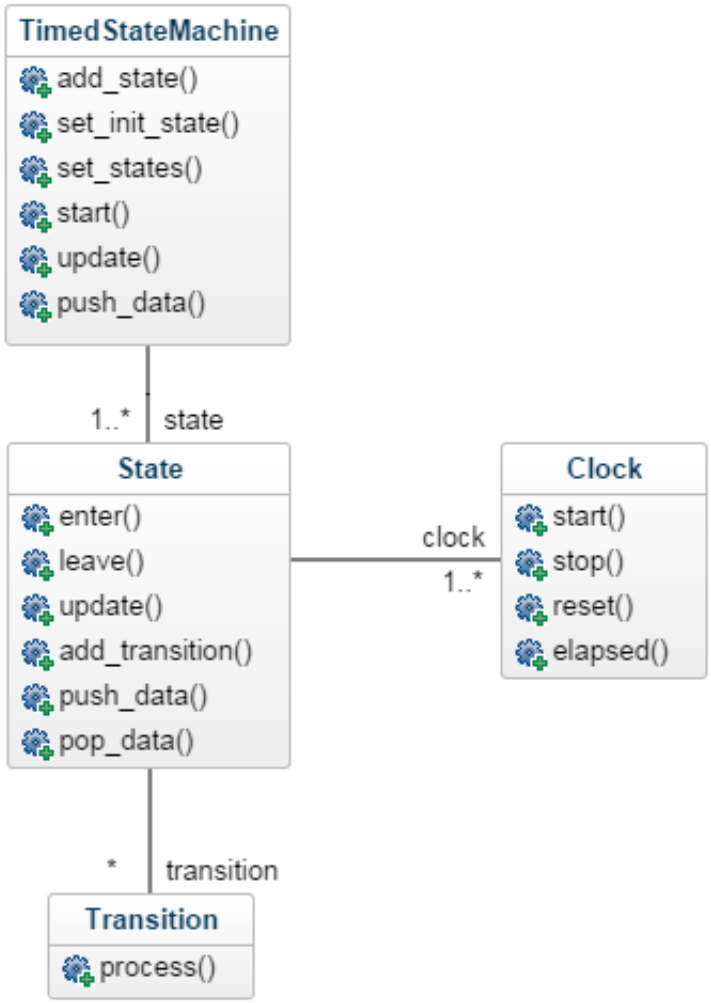

Figure 14: Class diagram of the timed state machine implementation. Each scenario is a timed state machine and each test is composed of one or many scenario.

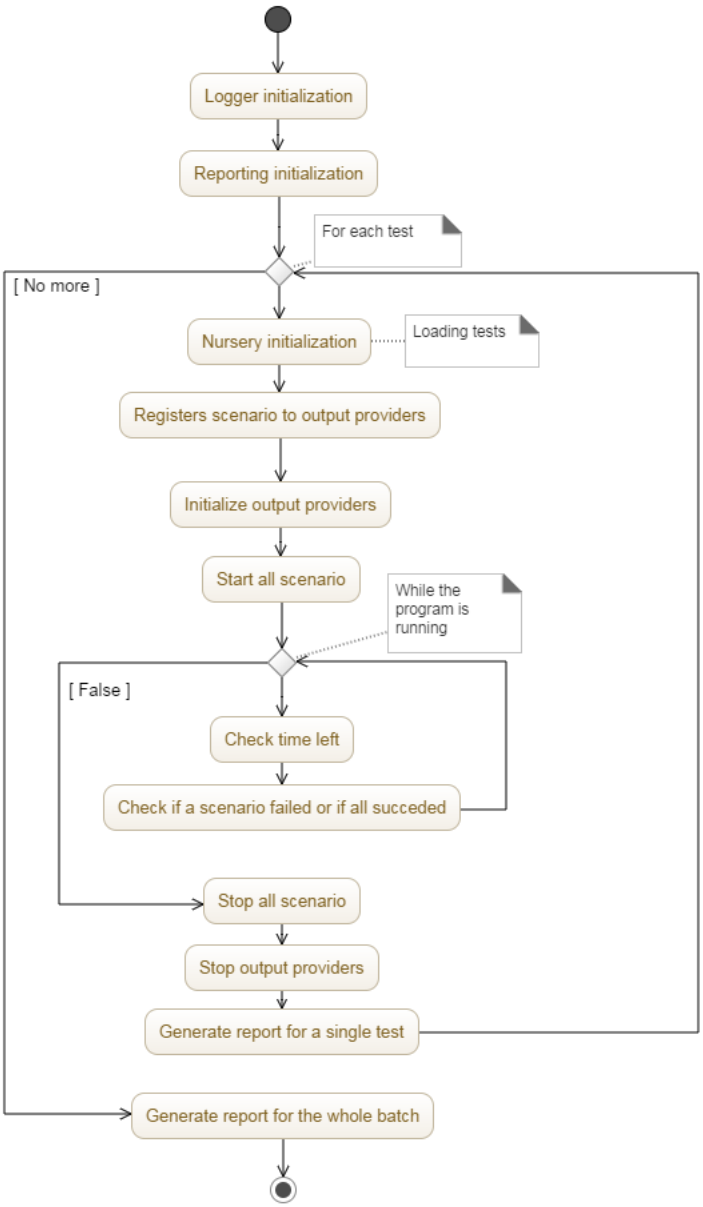

Figure 15: Activity diagram of the whole test processing. A batch of test is simply a list of one or more tests.

Listing 4: "Output provider configuration"

```
masterlist_op = [
    { "type": "WANESY", "name": "WanesyProvider" },
    { "type": "SERIAL_WANESY", "name": "SerialWanesyProvider" },
]
```

obtain the desired behavior. Most of the test used only one FSM because it proved difficult to conceive test thinking about multiple FSM at once. A typical case where it is useful to run multiple FSM at once is for timeout check. One of them checks continuously for timeout between data gathered by the output provider and the other one performs the test without having to manage this problem.

13.3 Command line tools

To run Nursery, the user can use a command line tool taking several inputs. Those inputs are the following:

- product: Type of hardware tested
- serial: RUG serial number to fetch informations with output providers. Needed only for RUG testing.
- password: Password, if any, for serial connection.
- log: Level of informations displayed in the log files, can be "critical", "warning", "info", "error" or "debug"
- quiet: Used to remove console outputs
- batch: Batch name to be performed. This override the **test** parameter
- test: Name of the test to be performed
- logpath: Specify where to put the resulting log file
- checkmaster: used to verify the configuration file. Checking if no errors where introduced

Listing 5: "Tests configuration"

```
l_test = [
    # Efficiency Vs Current
    {
        "name": "efficiency_vs_current",
        "l_product": ["battpack"],
        "description": "Test the efficiency against the current",
        "duration": 24*3600,
        "l_scenario": [
            {"name": "EfficiencyVersusCurrent", "settings": {}},
        ],
    },

    # CAT test
    {
        "name": "cat",
        "description": "CAT bullshit",
        "duration": 500,
        "l_scenario": [
            {"name": "ReplayBullshiter", "settings": {
                "filename": "cat.bullshit",
                "csv_delimiter": ",",
                "muted": True,
            }},
            {"name": "CATCheck", "settings": {}},
            {"name": "TimeoutCheck", "settings": {}},
        ],
    },

    # Charge Battery Pack
    {
        "name": "chargebattpack",
        "description": "Charge fully the battery pack",
        "duration": 7200,
        "l_scenario": [
            {"name": "ChargeBattPack", "settings": {}},
        ],
    },
]
```

Listing 6: "Batch of tests configuration"

```
l_batch = [
    {
        "name": "fingerprint",
        "l_test": [
            "serial",
            "symshow",
            "twctrl",
            "ifconfig",
            "qstat",
            "1970check",
            "gps",
            #"timeout",
        ]
    },
]
```

- report: Indicate whether or not we want to generate a report. This allows us to disable the reporting feature
- fqm: Shortcut to run a batch named "fingerprint" and generating reports for those tests

13.4 Web-administration

A web-administration tool was also developed. Allowing users to manage testing easily. All features covered by the command-line tool are available through the web-administration tool. We also added three others feature, the first one is the ability to browse all available tests and their documentation (figure 18) and the second one is the ability to see how a test was processed by displaying a automatically generated graph of the test and highlighting nodes of the graph depending on the current step selected in the trace file displayed next to the graph (figure 19). Finally, the ability to manage log files is a feature that is not covered by the command line tool (figure 17). All those features makes the web-administration tool a lot easier to use than the command line tool.

Listing 7: "Products configuration"

```
l_product_type = [
    {
        "type": "none",
        "description": """
            Applied to all hardware/software. Mainly used to disable re
        """,
        "reporting": "NoneReporting",
    },
    {
        "type": "rug1",
        "description": """
            Railster Universal Gateway 1
        """,
        "reporting": "RugReporting",
    },
    {
        "type": "ros",
        "description": """
            Railster-OS. Used for software testing
        """,
        "reporting": "NoneReporting",
    },
    {
        "type": "battpack",
        "description": """
            Testing for the battery pack
        """,
        "reporting": "NoneReporting",
    },
]
```

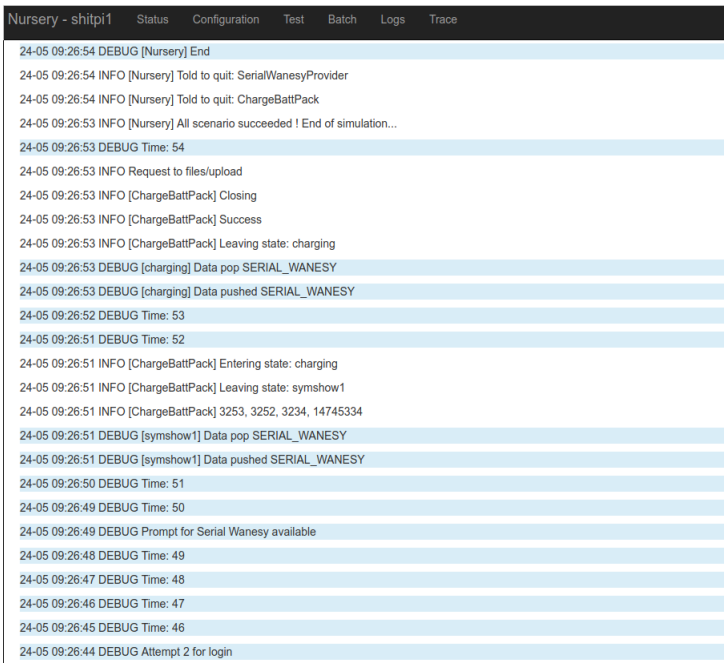

Figure 16: Main interface for the web-admin tool. Here we can see the last log file generated.

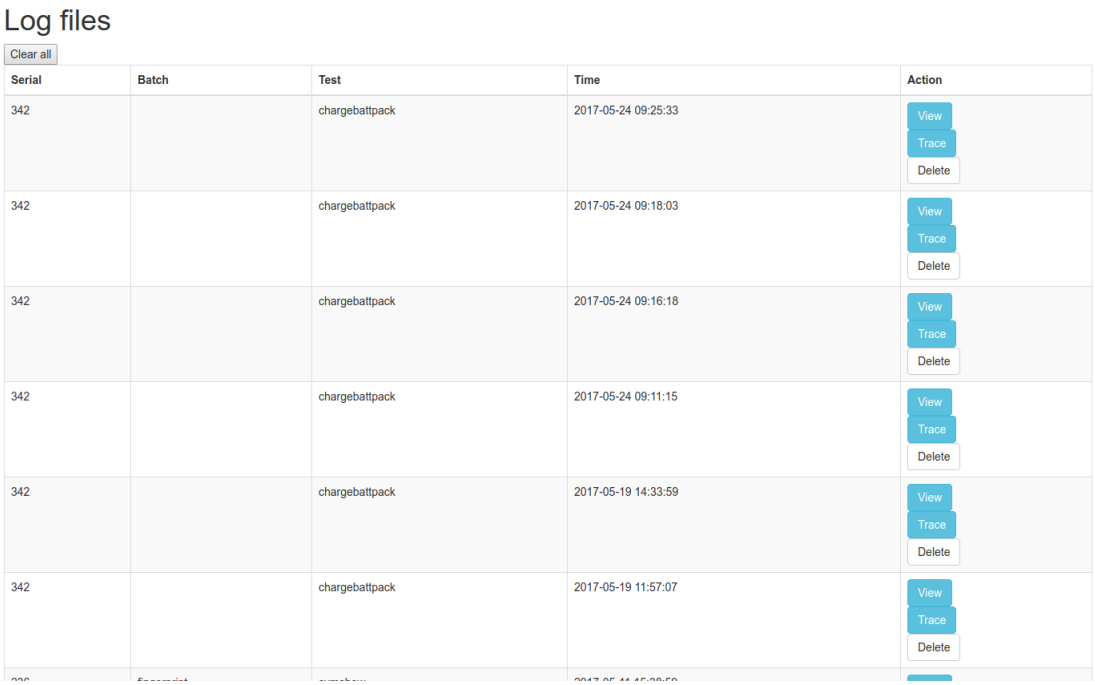

Figure 17: The log page displaying all logs generated and some informations about them.

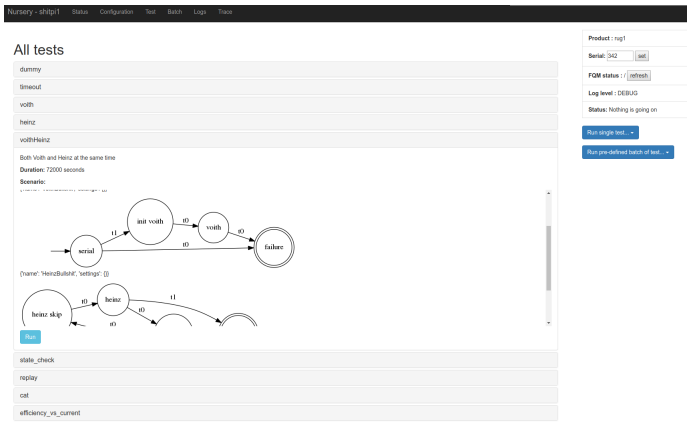

Figure 18: This page shows all the available documentation about each tests.

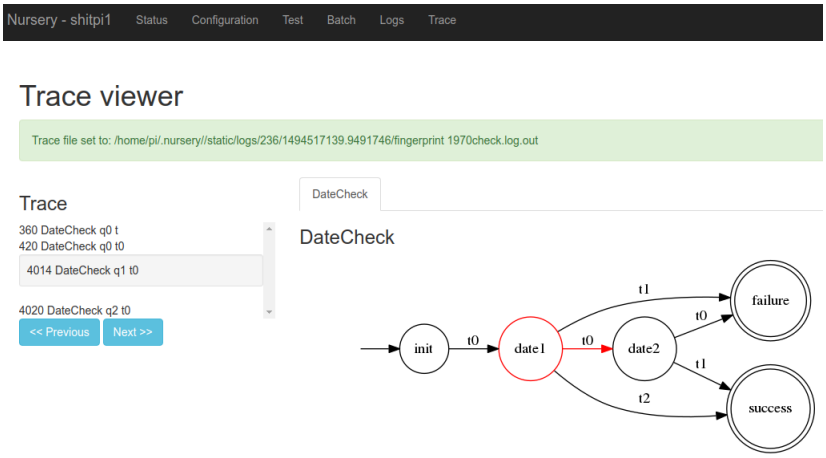

Figure 19: here we can use a trace generated by Nursery to review the test execution.

Part V
Conclusion

Putting the complexity as a limitation gave us a software that can run on any moder system. The simulation is hosted on RaspBerry Pi which are low end system. This allowed us to build a lot of Nursery system for a really cheap price, RaspBerry Pi costing around fifteen euros. We made around sixteen Nursery system separated into two areas. Height of them dedicated for RailsterOS verification and height of them used mainly for fingerprinting. So, using the work done in this paper, we are now able to fingerprint the faulty RUG way faster. We need only one guy to analyze up to height of them in parallel using the Nursery systems we just talked about. The same thing hold true for RailsterOS testing. We can now leave newly developed version of it running on Nursery for days and get a report of all the errors that happened afterwards instead of waiting for errors to happen into the production environment. The total count of line of code in Python is about 7000 and 17 tests where developed with more to come.

My personal analysis lead me to think that the success of Nursery is based on the following key point:

- Will it be used by other users ?
- Will it save time or make life easier for some people ?
- Can we gain money from it ?

Clearly, detecting fault earlier allows us to save money and time. No need to call back some hardware and less fault in production means less loss of money. We do not longer need to invest days of analyze for fingerprinting, all it takes is to plug the faulty RUG into Nursery and click a button. This could easily take up to 3 hours per RUG where now, we can do it in just one day of work for height RUG using more intensive testing cases ! Lastly, some guys of the hardware team and from the supply chain were introduced with Nursery. We showed them how to write test for Nursery and they loved the way things were made and how simple it was. Their test were done in approximately one hour. This lead us to think that, with proper formations

and documentation, other users could gladly uses Nursery.

I think that Nursery is a key assets that will become more and more used inside of Railnova. We even talked about using it for production lines in the future. But a lot of work is yet to be done on the accessibility aspect of it. Proper documentation needs to be written, some ideas are expected to ease a bit more the writing of tests like adding a generic state allowing us to create a transition from "any state" to one particular state without needing to repeat this transition to every other states and theres probably a lot more thing we didn't think about.

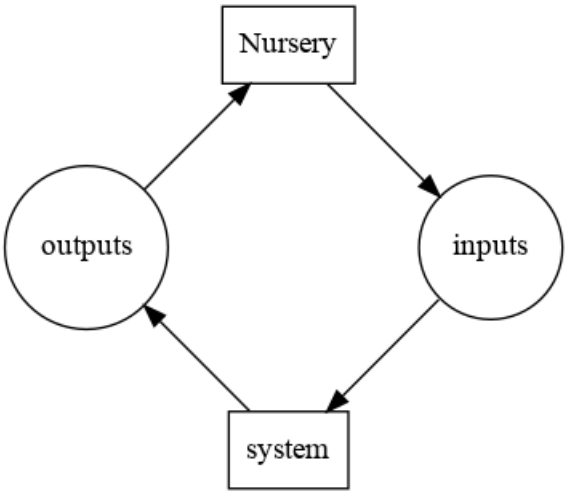

Figure 20: Inputs of the system are outputs of Nursery and vice/versa

A Abstraction of Inputs/Outputs

This part of the work is dedicated on describing how we managed to write simple piece of software that assure the link between Nursery and the real world/RUG.

A.1 About Inputs/Outputs

As we are working with a system that we feed with inputs and as we are ourselves feeded by the system's outputs, the concept of Inputs/Outputs (I/O) is a bit confusing. On the figure 20, you can visually see why it is ambiguous. So at this point, we'll simply consider that when we talk about inputs, we mean inputs of the system and when we talk about outputs, we talk of what the system produce.

A.2 Outputs

Let us start with the outputs. By outputs we mean all the data produced by the RUG or, more generally speaking, all data produced by the reviewed system. Those have been abstracted with a layer of object called output providers. Their goal is to retrieve outputs from various places and feed

them into Nursery as they go.

For example, we can imagine that we want to receive some data each time the system sends data to a server, each time the system produce a sound, etc. All of this is possible as we simply need to write a script that detects that and use some kind of API to feed them through Nursery.

A.2.1 How does it works

So, internally, each output provider defines a type of data it provides. For example, lets say we want to send data to nursery each time someone put a message on Twitter, we could name our data type: "TWITTER". Then, each test that is interested in data of type "TWITTER" will be subscribed to our output provider. It also allows us to ignore output providers for which no test are interested in. Practically speaking, the only output providers running for a given test will be those that provide a data type useful for the test.

Our provider is bundled with a mean of communication with all scenario that are subscribed to the data type it provide. With it, we can send data to every listener with one simple call. All the communication process is managed internally in Nursery. In our provider, we have a function that allows us to broadcast data to all listener. With this approach, we can write output providers for any type of data we can think of, all we have to do is defining the data type we provide and which output provider provides it in a configuration file and then implement this output provider, that's it, now any scenario can listen to this particular data type.

One limitation though is that every data type can be produced by at most one output provider. On the other hand, every output provider can produce as much data type we want if this becomes a requirement at some point.

A.2.2 Wanesy

The first data type we have develop is the WANESY data type. Data of type WANESY are triggered when we receive a new pack of data sent by RailsterOS to our server online. The method to retrieve them is simply to

poll a JSON generated list of message by HTTP and check timestamp of generated messages, when a newer timestamp appears, we get the message and send it into Nursery.

Another more effective way of doing this would be by direct connection to the database but this would probably introduce security issues and is not really required for our use case. Note that it is still feasible in future version of this work.

A.2.3 Log

This data type is more of Grey-Box testing as it requires a direct serial connection into the studied system so we assume that we are working on some kind of RUG and that we know where the log are located on it. Nevertheless, this data type ("LOG") is useful to provide more complex information wen fault arise and also help us to design more intrusive tests.

So even if they are a bit out of scope here, they do not invalidate the fact that our system can work on Black-Box testing.

To design those test, as we previously stated, we simply initiate a serial connection over the RUG and then download the file located in the log folder. Then, our script process it and send new lines into Nursery.

A.2.4 Miscellaneous

As we talked in sub-section A.2.3, we can initiate a serial communication with the RUG. Using this serial communication, we can send command to it in an Unix environment so it's easy for us to check a lot of properties for our tests needs. Of course, this is clearly white-box testing, we know how the system is supposed to work so we can ask it if some key file are in place, if all environment variables seems to be correct, do some statistics on some values to be sure they have correct values all along our tests and things like that.

Figure 21: Module used to manage the power status of a RUG

A.3 Inputs

Some of our test make use of external devices to produces inputs that are feeded to the RUG or to the hardware being tested. Those devices are, for example, a power module that can charge the battery of our RUG, thermic chamber that can manipulate the current temperature to stress test our hardware in extreme conditions and things like that.

We also have more direct inputs like the Ethernet connection, we want to be able to disconnect or re-connect the tested process or hardware.

A.3.1 Power Management

To control the power status, we need to send a command to a Wago module like the one of the figure 21. Those modules are connected via an Ethernet connection thus we need to open a TCP port to communicate with it and then send a command to select which power supply we want to control and what we want to do (power on or power off).

A.3.2 Connectivity

Controlling the connectivity is not something we did but this is easily done by using some advanced switch. We simply need to communicate with it and send commands to deactivate the Ethernet port to which the RUG is connected.

A.4 The generic way

Some techniques we mentioned, like in sub-section A.2.3 and sub-section A.2.4, use some kind of white-box testing. As one could expect, those techniques can only be used for one given device. We cannot initiate a serial connection with every process or hardware we want and all of them don't have the same Unix environment to dialog with.

But even the ones that applies to black-box testing can only be specific to some restraint set of process or hardware. Here, our techniques are all specific for RUG hardware. So every time we need to test a new kind of process or hardware, we will need to write some code to retrieve outputs and provides inputs for it. That's what we kept in mind all along this project by designing what we called Output Provider (O.P.).

A.4.1 Outputs

O.P. were designed to provides asynchronous capabilities to our tests. Polling I/O is a process that takes a lot of times compared to the execution time of our test. With O.P., we get the data when they are ready and we don't have to wait for them by freezing all the test. Two ways of retrieving data were implemented.

The most trivial way is simply to send data periodically to all test that needs it. The test knows he will receive data but he does not knows when. The test must be designed accordingly. This is useful to fetch data that are not fixed like a speed, the current time or the remaining fuel for example.

The second way is to ask for data from the test. The call is asynchronous so you simply notify the O.P. that you are ready to receive some data or that

you want it to fetch some data right now. This allows us to ask for one-time data like a serial number or other kind of data that are not supposed to change depending on the time.

So the first method is more for data you want to be able to monitor and the second method is more for data you will only need once.

A.4.2 Inputs

For inputs, we simply need some software to which we can send commands so we don't need so much abstraction as for the outputs. We are not in an asynchronous system. The way to go we adopted was simply to develop some software that accepts command and send inputs to whatever we want.

One example was a software we used that can control a thermionic chamber. With it, we were able to control the temperature in which our RUG was evolving. And of course, we can use it for other systems too, and we did it to test other pieces of hardware.

Another example, more specific, was a software used to control a device that could apply different electric loads to any device we connected with it. This was useful for some hardware test that had nothing to do with the RUG.

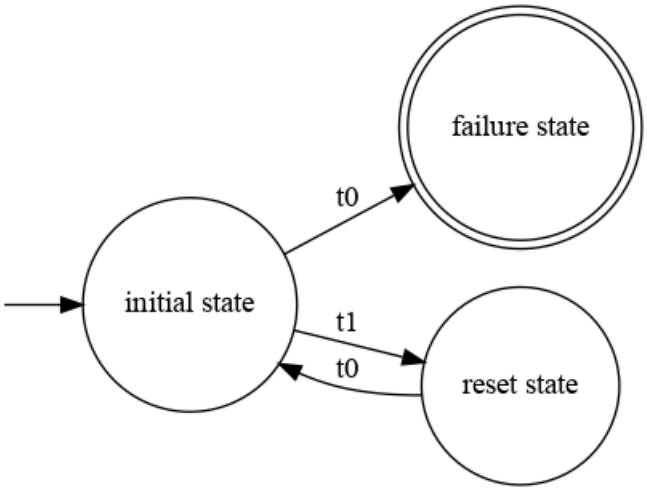

Figure 22: FSM for the timeout check

B Test cases

Here we will give explanation and details on some practical cases we had to deal with.

B.1 Timeout check

This test was designed to make sure that the RUG is sending informations to our servers. To do that, we need an O.P. that will fetch all messages received from the studied RUG and fetch them to our FSM. Our FSM will then check that data are received at least every 30 seconds.

The figure 22 shows how we designed this test. Transition are, of course, complex so we only show a simplified name for them. They are quite obvious given the rest of the diagram.

B.2 Date check

This test verify that the system uses the correct date. With an incorrect date set, the messages sent by it to the Railnova servers will be discarded resulting in a loss of communication from this device.

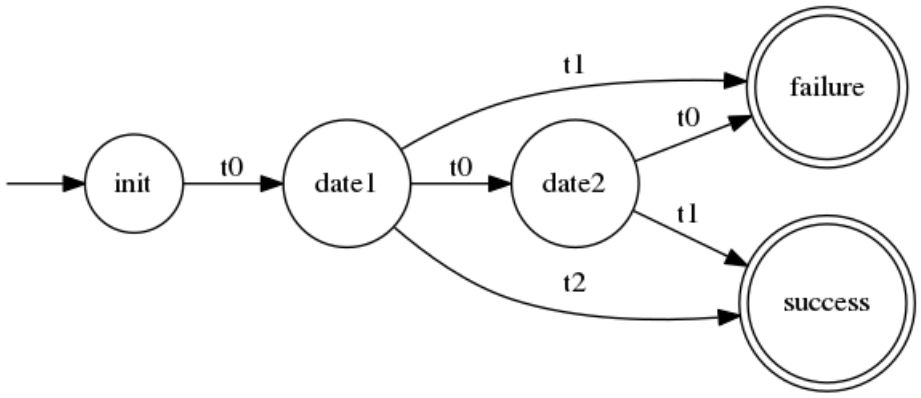

Figure 23: FSM for the date check

This test uses the serial connection and make uses of Unix commands directly on the system thus this is not a black-box testing processes.

The most complicated part here is to get the serial connection, a lot of troubles can arise while trying to log-in the system. Once we have a valid serial connection, we can simply issue a "date" command and parse the result. The date is updated through GPS/GSM antenna communication and this process can take up to an hour so if the date is not set right (if the date appears to be in 1970), we simply wait an hour and then, issue the test again.

1970 is the date you get when the timestamp is not set. This is because of how Unix and timestamp systems works.

B.3 Battery cycle

We made some test that where not related to the RUG. One of those was able to charge a battery and then discharge it. The goal of this was measure the electrical levels all along those steps to determine whether or not the battery is working.

As shown in the figure 24, we made several cycle of charge/discharge. This particular test was designed for a piece of hardware and not for a software. This is a sufficient proof to show that our system work as well for a software than it does for hardware.

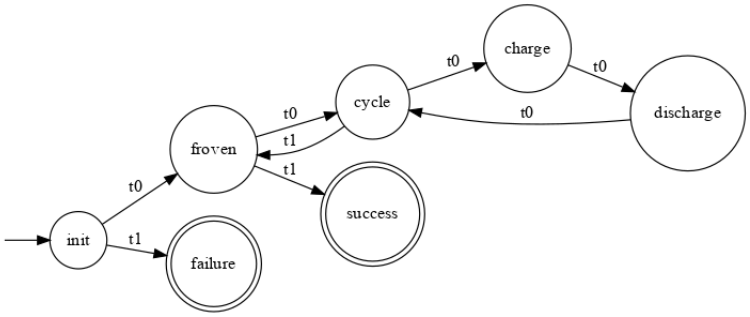

Figure 24: FSM for the battery cycle

References

[1] "Software development," https://en.wikipedia.org/wiki/Software_development, [Online; accessed 19-December-2016].

[2] H. Mooz and K. Forsberg, "4.4.3 a visual explanation of development methods and strategies including the waterfall, spiral, vee, vee+, and vee++ models," *INCOSE International Symposium*, vol. 11, no. 1, pp. 610–617, 2001. [Online]. Available: http://dx.doi.org/10.1002/j.2334-5837.2001.tb02348.x

[3] B. W. Boehm, "Software process management: Lessons learned from history," *ICSE '87 Proceedings of the 9th international conference on Software Engineering*, pp. 296–298, 1987.

[4] "What is scrum? an agile framework for completing complex projects," https://www.scrumalliance.org/why-scrum, [Online; accessed 29-April-2017].

[5] "Software verification and validation," https://en.wikipedia.org/wiki/Software_verification_and_validation, [Online; accessed 21-December-2016].

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

[7] A. W. Larry Brader, Howie Hilliker, "Testing for continuous delivery with visual studio 2012," https://msdn.microsoft.com/en-us/library/jj159344.aspx, [Online; accessed 31-October-2016].

[8] R. Patton, *Software Testing (2Nd Edition)*. Indianapolis, IN, USA: Sams, 2005.

[9] L. Williams, "Testing overview and black-box testing techniques," pp. 1–26, 2006.

[10] A. K. Dorota Huizinga, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, September 2007.

[11] ——, *Testing in Software Development*, ser. British Computer Society Monographs in Informatics. Cambridge University Press, February 1987.

[12] "Requirement based testing," https://www.ibm.com/support/knowledgecenter/SSJJ9R_5.0.2/ [Online; accessed 12-April-2017].

[13] "Requirements based testing process overview," pp. 1–18, 2009.

[14] "Finite-state machine," https://en.wikipedia.org/wiki/Finite-state_machine, [Online; accessed 22-March-2017].

[15] D. S. M. O. Rabin, "Finite automata and their decision problems," *IBM Journal*, pp. 1–12, April 1959.

[16] "Nondeterministic finite automaton," https://en.wikipedia.org/wiki/Nondeterministic_finite_a [Online; accessed 16-April-2017].

[17] "Alternate finite automaton," https://en.wikipedia.org/wiki/Alternating_finite_automaton, [Online; accessed 16-April-2017].

[18] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397594900108

[19] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, *UPPAAL — a tool suite for automatic verification of real-time systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 232–243. [Online]. Available: http://dx.doi.org/10.1007/BFb0020949

[20] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on Uppaal*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30080-9_7

[21] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 134–152, 1997. [Online]. Available: http://dx.doi.org/10.1007/s100090050010

[22] M. Kwiatkowska, G. Norman, and D. Parker, *PRISM 4.0: Verification of Probabilistic Real-Time Systems*. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2011, pp. 585–591. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_47

[23] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, *PRISM: A Tool for Automatic Verification of Probabilistic Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 441–444. [Online]. Available: http://dx.doi.org/10.1007/11691372_29

[24] "The economics of testing," http://www.riceconsulting.com/public_pdf/STBC-WM.pdf, [Online; accessed 12-April-2017].

[25] "Software tesing - economics," https://en.wikipedia.org/wiki/Software_testing#Economics, [Online; accessed 12-April-2017].

Glossary

AFA Alternating Finite Automaton. 29, 30, 54, 55

CTMC continuous-time Markov Chain. 35

daemons Sub-programs running inside RailsterOS adding functionalities to it. 39

DFA Deterministic Finite Automaton. 29

DTMC Discrete-Time Markov chain. 35

FQM Fleet Quality Management. 44

FSM Finite State Machine. 8, 9, 24–29, 31, 36, 37, 52–56, 60, 63, 78–80

I/O Inputs/Outputs. 72, 76

MDP Markov decision process. 35

NFA Non-Deterministic Finite Automaton. 28, 29

O.P. Output Provider. 76, 78

PA Probabilistic Automaton. 35

PTA Probabilistic Timed Automaton. 35

RailsterOS A set of sub-program running on the RUG defining the behavior of it. 38–40, 42, 50, 73, 84

RUG Railster Universal Gateway. 38–42, 44, 50, 63, 72, 74–79, 84

UI User Interface. 55

Faculté des Sciences
Département d'Informatique

UNIVERSITE LIBRE DE BRUXELLES,
UNIVERSITE D'EUROPE

ACADÉMIE UNIVERSITAIRE
WALLONIE-BRUXELLES

Mémoire présenté en vue de l'obtention

Du diplôme du Master en Sciences informatiques

Année académique 2016 - 2017