University of São Paulo
Institute of Mathematics and Statistics
Bachelor in Computer Science

Rafael Mota Gregorut

Synthesising formal properties from statechart test cases

São Paulo
December 2015

Synthesising formal properties from statechart test cases

Final monograph for the course
MAC0499 – Final Graduation Project.

Supervisor: Ana Cristina Vieira de Melo

São Paulo
December 2015

Abstract

This project explores the theme of software quality through the perspectives of testing and formal methods. Testing is a crucial activity in the software development cycle to provide the reliability of systems. In order to validate the system requirements, the test cases are created based on the requirements contained in the specification. Also concerned with systems quality assurance, formal methods provides several techniques and tools to be used during specification, design and verification phases. Statecharts are a particular kind of formal specification based on finite state machines mostly used to model reactive system behaviours. Testing and formal methods can be seen as complementary approaches to assure software quality. Testing can show the presence of errors, while formal verification, such as model checking, can prove their absence. These two techniques are very costly in software development and each one requires that developers are specially trained to each activity. For the software formal verification, one needs to define properties in a certain specification language, which requires strong mathematical background. On the other hand, generating test case from specifications is time consuming and very error prone if it is manually executed. To facilitate the creation of test cases and the specification of formal properties by developers, two techniques were studied and implemented. First, we automatically generate test cases for a given Statechart model. Second, we automatically synthesise properties based on the test cases previously obtained. This monograph was prepared for the course MAC0499 - Final Graduation Project, at the Institute of Mathematics and Statistics of the University of São Paulo.

Keywords: Statecharts, Testing, Test cases, Test case mining, Formal properties

Contents

Chapter 1

Introduction

Testing is one of the most used techniques to assure the quality of systems in software engineering and it typically consumes from 40% up to 50% of the software development effort [18]. As presented in [2], the goal of testing depends on the maturity level of the organization that is executing the tests: from debugging code, at a more inexperienced level, to a mental discipline that helps all professionals in the software industry to increase quality, at a higher level. In the literature, the testing techniques are categorized in structural (white-box) and functional (black-box), tests. The first category considers the code as the source for test cases and contributes mainly to code maintenance. Functional tests make usage of the specified requirements in order to create the test cases and are useful to validate the observed behaviour of the system against what was expected in the requisites.

Thus, it is notable that the requirement specification is an important source of material not only for the development of the code, but for testing as well. Statecharts are a type of formal software specification based on finite states machines (FSM) specially used in complex systems modeling, such as reactive systems[15]. They contain states, transitions and input events similar to an automaton, but offer resources to model hierarchy of states, concurrency and communication between process.

Formal methods are a set of techniques and tools which supports not only rigorous specification, but also design and verification of computer systems [11]. These techniques are mostly used in critical components of safety critical systems to assure the quality of the software. But they can also be applied to requirements and high-level designs where most of the details are abstracted away and to the verification process as well, using as much automation as possible[24].

Formal methods and testing can be considered complementary techniques in software engineering, both with the target to reduce the number of errors and increase the reliability of the system[6]. Even though testing is the activity that is most commonly used in industry to assure the software quality, it cannot guarantee the absence of bugs in the code, as stated by Dijkstra [7]. Formal approaches, such as formal verification, on the other hand, can prove the correctness of the code regarding a specification. The model checking technique, for instance, proves that certain user defined properties are true for a given model of the system.

This project proposes a study on the set of test cases generated from specifications in order to guide formal properties definitions. First, a technique to automatically generate test cases from statechart specifications is presented. Then, all test cases are observed and a technique to synthesize formal properties (written in linear temporal logic[16, 30]) from the acquired test cases is proposed.

1.1 Motivation

In the model driven development, test cases to validate systems can be generated from a model that represents how the system must behave. For instance, if the services to be provided by a system are described in Statecharts models, one can automatically generate test cases for programs from such models. However, properties to be proved in a system are not synthesized from such models. In general, developers take the system specification and observe its restrictions to then define which properties should be satisfied. However, creating such properties requires a strong mathematical background and translating system requirements to formal properties is not trivial process[13].

1.2 Goals

- Implement a technique to automatically generate test cases based on statechart specifications
- Analyse a set of test cases and extract the events, and sequences of events, that are found in such whole set
- Using the previous information, implement a technique to synthesize properties that can be used in formal verification
- Develop tools to help in these tasks

1.3 Organization

In chapter 2, we present the concepts studied to make this project possible. In chapter 3, we describe a technique implemented to automatically generate test cases from statechart models. In chapter 4, we propose a technique to synthesize formal properties based on the previously obtained test cases. Chapter 5 presents a demonstration of the implemented tools. Our conclusions regarding this project can be found in chapter 6. In chapter 7, subjective aspects of the project that were relevant to the author are presented. Finally, the syntax of linear temporal logic, used in the generated formal properties, can be found in apedix A.

Chapter 2

Concepts

2.1 Statecharts and Automata

A software specification is a reference document which contains the requirements the program should satisfy. It can also be understood as a model of how the system should behave. A specification can be developed with natural language, with user cases for example, or using formal software engineering techniques.

Statechats are a type of formal software specification based on finite states machines (FSM) specially used in complex systems modelling, such as reactive systems, and were defined in [15]. Similar to an automaton, a statechart has sets of states, transitions and input events that cause change of state. However, a statechart has additional features: orthogonality, hierarchy, broadcasting, guard conditions and history. Due to these additional features, statecharts have been used in industry to specify mission critical systems.

2.1.1 Nondeterministic finite automata

To understand how statecharts can be used to specify systems, we must first have the basic automata background.

Definition[17]: A nondeterministic finite automaton is a quintuple $M = (K, \Sigma, \Delta, s, F)$, where:

- K is the set of states
- Σ is the input alphabet
- Δ is the transition relation, a subset of $K \times (\Sigma \cup e) \times K$, where e is the empty string
- $s \in K$ is the initial state
- $F \subseteq K$ is the set of final states

Each triple $(q, a, p) \in \Delta$ is a transition of M. If M is currently in state q and the next input is a, then M may follow any transition of the form (q, a, p) from state q to state p. In nondeterministic finite automaton, it also possible to have transitions labelled with the empty string e. In this case, when such transition is taken, no input is read and the transition still occurs. For example, if from state q there is a transition (q, e, p), we can de redirected to state p even though no input was received.

A configuration of M is an element of K^* and the relation $\vdash$ (yields one step) between two configurations is defined as: $(q, w) \vdash (q', w') \Leftrightarrow$ there is a $u \in \Sigma \cup e$ such that $w = uw'$ and $(q, u, q') \in \Delta$.

For further information and formalisms regarding finite automata the reader is referred to [17].

A nondeterministic finite automaton example

Find below an example of a nondeterministic finite automaton extracted from [17].

- $K = \{q_0, q_1, q_2\}$ (the set of states)
- $\Sigma = \{a, b\}$ (the alphabet used)
- $\Delta = \{(q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_0), (q_2, e, q_0)\}$ (the transitions)
- $s = q_0$ (the initial state)
- $F = \{q_0\}$ (the set of final states)

The corresponding graphical representation is given in Figure [**?**].

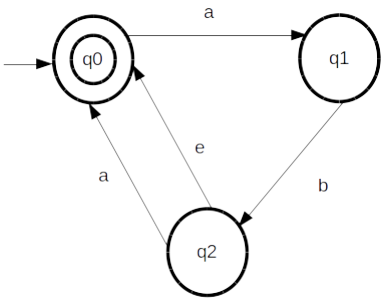


Figure 2.1: *A nondeterministic finite automaton that accpets language* $L = (ab \cup aba)^*$.

Consider the case in which we have the automaton represented in Figure 2.1 and the input ab. We start at state q_0, the initial state, and take as input the symbol a. Since there is a transition (q_0, a, q_1) we are redirected to state q_1. Next, we receive as input the symbol b and analogously, we are redirected to state q_2. At this point, there is no input left to read, but since there is a transition (q_2, e, q_1) we are still able to proceed to state q_0 without receiving any input. Hence, we finished the input processing in q_0 and, because $q_0 \in F$, we conclude that ab is a valid input for the described automaton.

2.1.2 Statechart models

A statechart model can be considered as an extension of finite state machines. The syntax of statechart is defined over the set of states, transitions, events, actions, conditions, expressions and labels [15]. In short terms:

- Transitions are the relations between states
- Events are the input that might cause a transition to happen
- Actions are generally triggered when a transition occurs

- Conditions are boolean verifications added to a transition in order to restrict the occurrence of that transition
- Expressions are composed of variables and algebraic operations over them
- A label is a pair made of an event and an action that is used to name a transition

Furthermore, it is worth to note that statechart models have additional features, described afterwards in this chapter, that do not appear in an ordinary finite state machine. These extra resources are useful, for instance, to model concurrency and different abstraction levels in a more practical way.

Figure 2.2 displays a statechart model for a clock [22]. This clock contains two buttons, one triggers the event *mode* and the other the event *set*. All configuration in the clock should be done through these two events. Time can be displayed in hours or in minutes, as the user wishes. The settings should allow the user to adjust the value of the hours and the value of the minutes. Besides, the clock should provide an alarm feature with the following behaviour: if the alarm is activated, then it will ring for five seconds every minute. The alarm activation and deactivation should be configurable too.

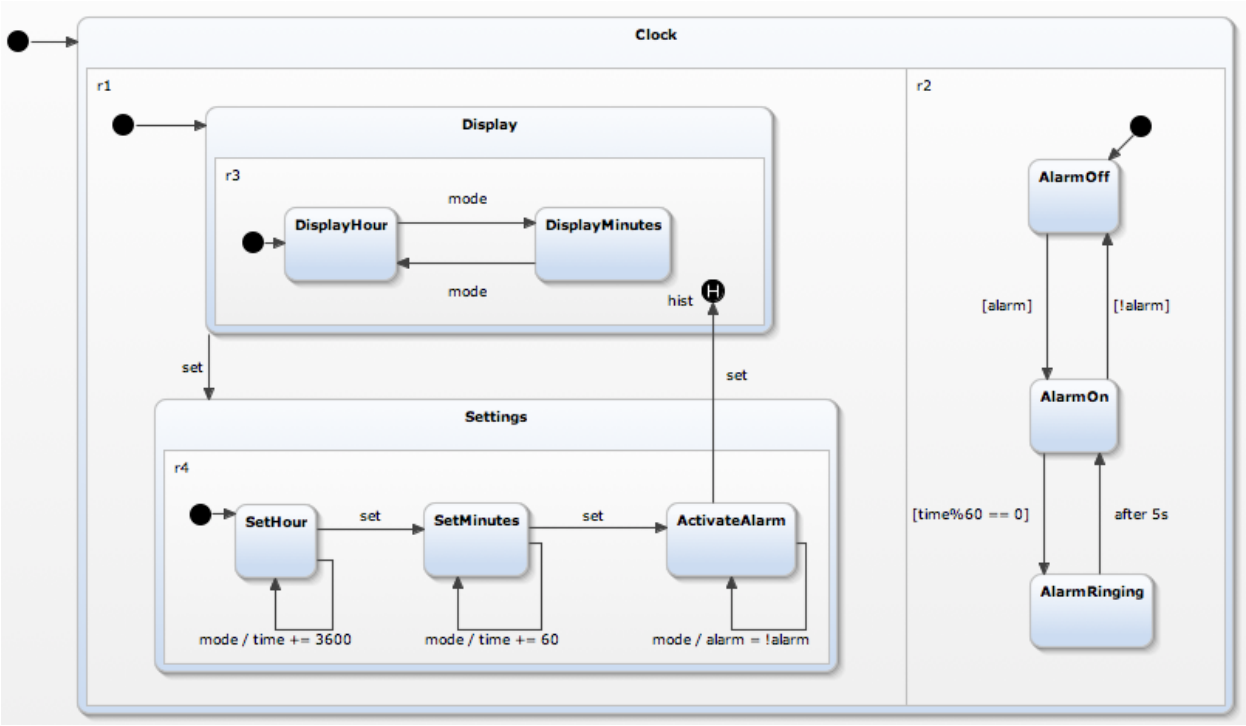


Figure 2.2: *Statechart that models a clock obtained from [22]*

Orthogonality

More than one state can be active at the same time in a statechart, this is called orthogonality. It can be used to model concurrent and parallel situations. The set of active states in a certain moment is called configuration. In Figure 2.2, we have parallel regions inside the state *Clock* ($r1$ and $r2$ are defined). This means that states *Display* and $AlarmOff$ can be active simultaneously. Once we are in the *Clock* state, we are redirected to its subregions $r1$ and $r2$. In $r1$, we proceed to state *Display* and, in $r2$, we go to state $AlarmOff$. At this moment, both states *Display* and $AlarmOff$ are active. If the *set* event occurs, we are redirected to state *Settings* (in $r1$), but $AlarmOff$ remains active at the same time.

Hierarchy

It is possible that a state contains other states, called substates, and internal transitions, increasing the abstraction and encapsulation level of the model. In Figure 2.2, we have that $Display, Settings, AlarmOff, AlarmOn$ and $AlarmRinging$ are substates of $Clock$. $Display$ also contains the substates: $DisplayHour, DisplayMinutes$ and $hist$. And the states $SetHour, SetMinutes$ and $ActivateAlarm$ are inside state $Settings$. We can have as many nested states as required, giving distinct abstraction views of the system.

Guard conditions

In a transition, a guard condition is always verified before the change of states take place. If the condition is satisfied (if its expression returns true), the transition will be engaged. Otherwise, that transition is not allowed to happen. A expression in a guard condition involve variables and operations. In Figure 2.2, the transition from $AlarmOff$ to $AlarmOn$ is guarded by the condition $[alarm]$, meaning that the transition will only happen if the value of the variable $alarm$ is $true$.

Broadcasting

A transition might have an action that is triggered once the transition is performed. An action may cause another transitions to happen, which is called broadcasting. This feature allows that chain reactions to occur in a statechart model. Considering Figure 2.2, if we are currently in states $ActivateAlarm$ and $AlarmOff$ and the $mode$ event occurs, we will have the following situation: the transition will be execute and we will stay at state $ActivateAlarm$, the value of variable $alarm$ will be changed to its opposite (let's supposed it was $false$, so now it will be $true$) and we will also go to state $AlarmOn$ since the condition $[alarm]$ guarding the transition between these last two state is satisfied. Therefore, not only the transition starting at $ActivateAlarm$ happened when $mode$ occurred, but also the transition between $AlarmOff$ and $AlarmOn$. The change of the value of variable $alarm$ was broadcasted to the whole statechart and a chain reaction happened.

History

A statechart is capable of remembering previously visited states by accessing the history state. Consider Figure 2.2, suppose we are in state $DisplayMinutes$ and event set happened three times in a row. So, we went to $SetHour$, and then $SetMinutes$ and now we are at $ActivateAlarm$. If there is another occurrence of set event, we will be directed to the history state $hist$. Since the last active substate in $Display$ was $DisplayMinutes$, we will actually be directed to $DisplayMinutes$.

2.1.3 A statechart and its corresponding automaton

It is possible to obtain an automaton from a statechart by flattening the statechart. The hierarchy and concurrency need to be eliminated. In addition, statechart elements such as guard conditions and actions are not present in an automaton.

The following examples were extracted from [15]. In Figure 2.3 we have a statechart and in Figure 2.4 its flat version that would correspond to an automaton.

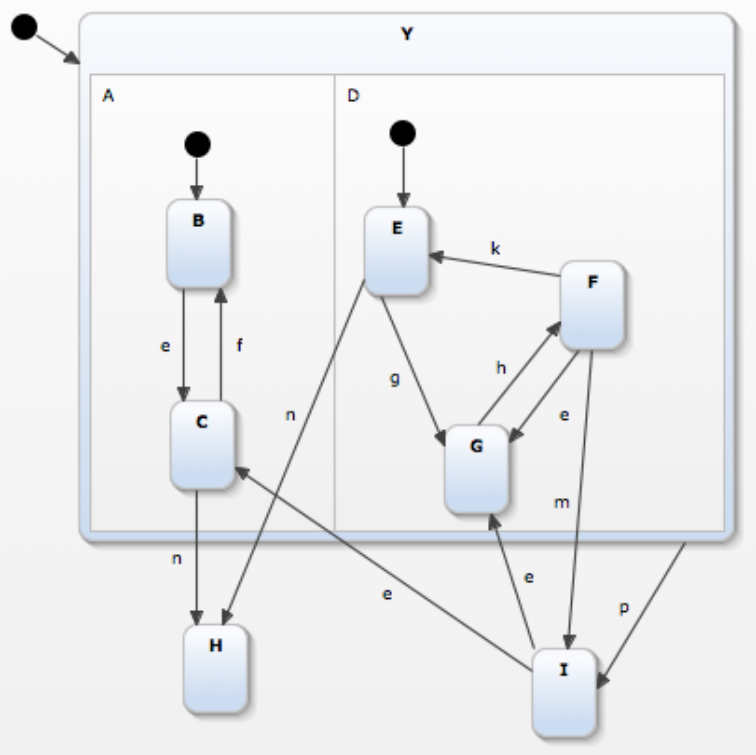


Figure 2.3: *A statechart model with hierarchy and concurrent regions*

Hierarchy removal

To eliminate hierarchy, we must handle transitions that leave and enter composed states. Consider a transition $t = (s_1, l, s_2)$ that is labelled with event l, redirects to state s_2 and leaves from state s_1, which is composed by substates $q_1, ..., q_n$. We need to create n new transitions such that $t_i = (q_i, l, s_2), 1 \leq i \leq n$. In other words, we are transferring transitions that leaves the composed state s_1 to its subtates $q_1, ..., q_n$.

We should treat transitions that enter composed states too. Consider a transition $t = (s_1, l, s_2)$ that leaves from state s_1, is labelled with event l and arrives in a composed state s_2, which contains substates $q_1, ..., q_n$ such that q_1 is the initial substate. We must create a new transition $t' = (s_1, l, q_1)$. This means that the transition t' skips the composed state s_2 and redirects directly to the initial substate q_1.

Finally, to eliminate hierarchy, we should remove composed states and keep the states located in the lowest level of hierarchy of the statechart.

Orthogonality removal

To remove orthogonality, we need to do the Cartesian product of the states in each concurrent region. Such operation causes a reasonable number of state and transition explosions in the automaton, if the original statechart model has concurrent states.

Consider two concurrent regions $r1$, with states $s_1, ..., s_n$, and $r2$, with states $q_1, ..., q_m$. For each $1 \leq i \leq n$, and for each $1 \leq j \leq m$, we should create a state u_{ij} in the automaton. Transitions $t = (s_i, l, s_k)$ should be converted into $t' = (u_{ij}, l, u_{kj})$ and transitions $v = (q_j, l, q_k)$ should be converted into $v' = (u_{ij}, l, u_{ik})$.

Note that, in the automaton, the initial state is the product of the initial states from each parallel region in the statechart. Finally, the states and transitions in the original statechart can be removed.

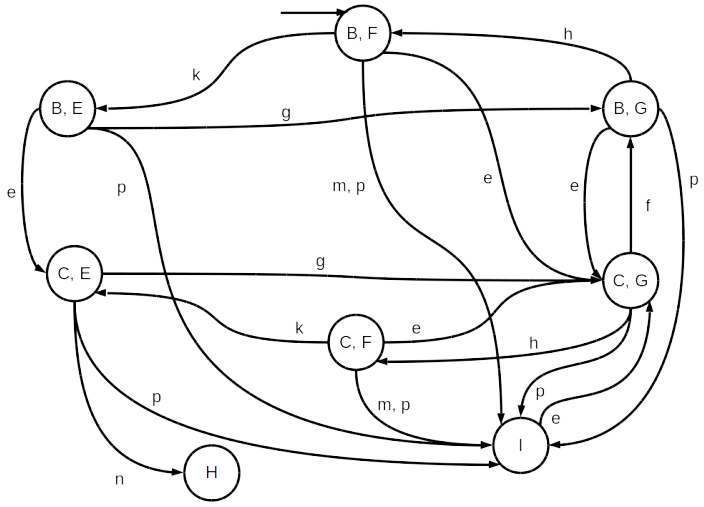


Figure 2.4: *The statechart from 2.3 after flattening to the corresponding automaton*

It is important to notice that, in the statechart, we can use guard transitions, broadcasting and history, enriching the model with more information. However, none of these features are available in the automaton model.

2.2 Software testing

2.2.1 Testing goals

Testing is one of the most important means of assessing the software quality and it typically consumes from 40% up to 50% of the software development effort [18]. In an organization, the goals of testing depends on the level of maturity of the team [2]. At a more naive approach, testing could be viewed as debugging code. A further step would be to consider testing as a process to make sure that a software does what it is required to do. But, in this case, if a team runs a test suite and every test case passes it, should the team consider that the software is correct? Are the test cases not enough? Do they guarantee the software quality? How does the team decide when to stop testing?

At a different perspective, testing can be understood as a process of finding errors. Therefore a successful test case would be the one that indicated an error in the system [25]. Altough discovering unexpected results and behaviours is a valid goal, it might put tester and developers in an adversarial relationship, which certainly damages the team interaction.

Testing shows the presence, but not the absence of errors. Hence, realizing that when executing a software we are under some risk that may have unimportant or great consequences leads to another way to see the intention of the test process: to reduce the risk of using the program, an effort that should be performed by both testers and developers.

Thus, testing can be seen as a mental discipline that helps all professionals in the software industry to increase quality [2]. It can be considered a process not only to detect bugs, but also to prevent them [4]. The whole software development process could benefit from that thinking: design and specification would be more clear and precise and the implementation would have fewer errors and would be more easily validated, for example.

2.2.2 The process of testing

Since testing is a time consuming activity, the creation of test cases can be done during each stage of the development process, even though their execution will only be possible after some part of the code is implemented.

V-Model

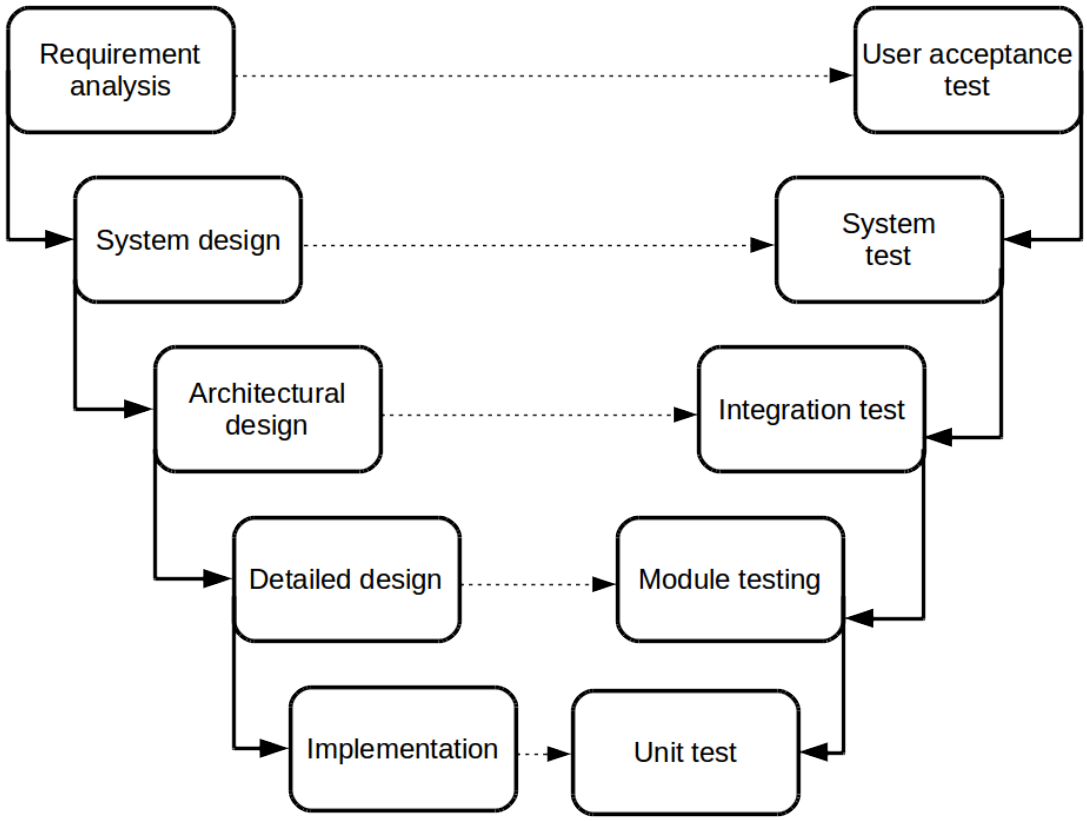


Figure 2.5: *The V-model*

The V-model shown in Figure 2.5 associates each level of testing to a different phase in the development process. This model is typically viewed as an extension of the waterfall methodology, but it does not imply the waterfall approach, since the synthesis and analysis activities can be applied to any development process [2].

- Requirement analysis phase: Customer's needs are registered. **User acceptance tests (UAT)** are constructed to check whether the delivered system meets the customer's requirements.
- System design phase: Technical team analyses the captured requirements, studies the possible implementations and documents a technical specification. **System tests** are designed at this stage assuming that each component works accordingly and checks for the system to work as a whole.
- Architectural design phase: Specify interface relationships, dependencies, structure and behaviour of subsystems. **Integration tests** are developed, with the assumption that

each module works correctly, in order to verify all interfaces and communication among subsystems.

- Detailed design phase: A low-level design is done to divide the system in smaller pieces, each of them with the corresponding behaviour specified so that the programmer can code. **Module testing** is designed to check each module individually.
- Implementation phase: Code is actually produced. **Unit tests** are designed to test if every smallest unit of code, such as a method, can work correctly when isolated.

2.2.3 Functional and Structural tests

Testing techniques can be divided into functional and structural categories.

Functional technique (black box)

The software behaviour described in the specification is considered to create the test cases. It is necessary to study the behaviour, develop the corresponding test cases, submit the system to the test cases and analyse the results observed comparing them to what was expected according to the description in the specification[20]:

- **Equivalence classes partitioning**

  Input data are partitioned into valid and invalid classes according to the conditions specified. The test cases are created based on each class by selecting an element in each class as a representative of the whole class. Using this criterion, the test cases can be executed systematically according to the requirements.

- **Analysis of the boundary value**

  Similar to the previous criterion, but the selection of the representative is done based on the classes' boundary. For that reason, the conditions associated with the limit values are exercised more strictly.

Since many specifications are written in a descriptive fashion, the test cases developed using the functional technique can be very informal and not rigorous enough, which makes it difficult to automate their execution and human intervention becomes necessary. However, this technique only requires that requirements, input and corresponding output are identified. Thus, it can be applied during several testing phases, such as integration and system testing.

Structural technique (white box)

The structure of the code implementation is considered to create the test cases in this category. In this technique, the program is represented by an oriented flow control graph, in which each vertex corresponds to a block of code (a statement in the code for instance) and each edge corresponds to a transition between blocks. The criteria related to this technique are concerned with the coverage of the program graph, such as [2]:

- **Node coverage**

  Every reachable node in the graph must be exercised by the test set. Node coverage is implemented by many testing tools, most often in the form of statement coverage

- **Edge coverage**

  Every edge in the graph must be tested in the test set.

- **Complete path coverage**

  All paths in the graph must be tested by the test set. This criterion is infeasible if there are cycles in the graph due to the infinity number of paths.

- **Prime path coverage**

  A path is a prime path if it is a simple path and it does not appear as a proper sub-path of any other simple path. In this criterion, all prime paths should be tested.

- **Specified Path Coverage**

  A specific set S of paths is given and the test set must exercise all paths in S. This situation might occur if the use case scenarios provided by the customer are converted to paths in the graph and the team wishes to test them.

Tests obtained via the structural technique contribute specially to code maintenance and increase the reliability of the implementation. But, they do not represent a way to validate the system against the customer's requirements, since the specified requirements are not considered in their design.

2.3 Test cases

In this section, we will be concerned with test case creation from the functional perspective. The structure of the code will not be analysed. Instead, the specifications with the requirements are going to be the source to derive the test cases.

2.3.1 Desiging test cases

For simple programs, a test case is a pair comprising the software input and the expected output. But, for more complex software, such as web applications or reactive systems, a test case can be a series os consecutive steps or events and their expected consequence.

Given a specification, a test engineer can systematically design test cases: for each requirement defined, use the equivalence class partitioning technique described in 2.2.3 to select the input representatives; enumerate the steps necessary to activate the requirement; and add the expected outputs. Note that preconditions to each step can be added as well, specially to guarantee that steps are not performed out of order and that the whole flow will be completely executed and tested.

A good design of test cases is essential to the entire testing activity since it will guide testers through the scenarios they should execute and check. The created test cases will determine which requirements will be tested and how they will be tested. Besides, the whole test plan tends to be planned around the test cases, considering the amount of cases and their complexity to estimate deadlines and necessary resources. Moreover, managers usually use test cases and their execution rate to give the client feedback about testing progress.

Furthermore, when defects are identified, they can be associated to the test case that caused their detection. Since each test case is associated with a requirement, the team can easily keep track of how many defects each requirement has, which scenarios are more problematic and which critical bugs are associated with them.

Writing test cases manually is not a trivial task. The engineer must read through all the specification, identify requirements and the many scenarios in which that can be executed. It requires experience and time to build test cases that can better discover defects.

Use cases vs Test cases

Let us consider a web application defined as follows: A web application requires that users sign in to access their personal homepage. To do so, the user should first access the regular homepage and click on the button labelled *sign in*. Then, the user will be redirected to the sign in page, which contains a form with fields *email* and *password*. Once the form is filled in and another button *sign in* is clicked, the system should validate the user's email and password. If the credentials are valid, them the user will be redirected to their personal homepage. Otherwise, an error message will be displayed and the user will remain in the form page.

For the given problem, we can have an use case of signing in a website:
Use case: Website sign in
Actor: Registered user
Preconditions: User has a register in the website and is able to access the home page.
Basic flow:

1 System loads the home page of the website, which contains a link to sign in

2 The user clicks on the sign in link and is redirected to the sign in form

3 The user fills in the sign in form and clicks on button 'sign in'

4 The user is able to sign in and their personal home page is displayed

Alternative flows:

a Sign in fails

1 System loads the home page of the website, which contains a link to sign in

2 The user clicks on the sign in link and is redirected to the sign in form

3 The user fills in the sign in form with invalid information and clicks on button 'sign in'

4 The user is not able to sign in and system displays an error message

More detailed use cases provide sequences of steps to perform flows related to a particular requirement. Each step might contain the description of system behaviour and user actions, including preconditions and actors that take part in the use case. Besides that, one use case might describe more than one scenario for the requirement by extending the basic flow.

To provide the corresponding test cases, however, each step in the test case must specify a single action and its corresponding consequences to the system. We can add preconditions to give a more precise definition of each step in a test case, even if the precondition only says that the previous step was executed successfully. In addition, it is not possible to have more than one flow in one test case, otherwise the tester would have no clear direction during the test run. Therefore, we need to create test cases for each flow separately.

To test the scenario in which the user is able to sign in successfully, one can consider the following test case:

Step	Precondition	Description	Expected result
1	Credentials are valid and user is able to access home page	Access home page	Home page is loaded
2	Step 1 was successful	Click on Sing in link	Sign in page is displayed with the sign in form
3	Step 2 was successful	Check the sign in form	It should contain fields login, password and a button sign in
4	Step 3 was successful	Type the user's email in field login	Login field will hold the data
5	Step 4 was successful	Type the user's password in password field	Password field will hold the data
6	Step 5 was successful	Click on button sign in of the form	Credentials should be successfully validated by the system
7	Step 6 was successful	Check the page that was loaded	It should the user's personal home page

It is also interesting to project test cases for negative scenarios, in which the program should handle an incorrect action done by the user. A test engineer could consider using the following test case to test the negative scenario for the sign in requirement:

Step	Precondition	Description	Expected result
1	Credentials are invalid and user is able to access home page	Access home page	Home page is loaded
2	Step 1 was successful	Click on Sing in link	Sign in page is displayed with the sign in form
3	Step 2 was successful	Check the sign in form	It should contain fields login, password and a button sign in
4	Step 3 was successful	Type the user's email in field login	Login field will hold the data
5	Step 4 was successful	Type the user's password in password field	Password field will hold the data
6	Step 5 was successful	Click on button sign in of the form	Credentials should not be validated and error message should be displayed
7	Step 6 was successful	Check the error message displayed	It should be "Wrong email or password."

Thus, an use case serves also as a guide for the implementation process used by developers. It is a way to understand how the requirement that is being coded is going to be used. An use case will not be executed directly. Test cases are used to direct tester regarding the precise actions that should be performed in each flow and are a way to detect errors. They are directly executed during the test phase.

Use cases are a great source for the creation of test cases because they detail main flows and contain action steps. However, they are different in structure and purpose; therefore, one cannot replace the other.

2.3.2 Automatic generation of test cases

A model, a statechart for instance, can also be used to specify certain scenarios and requirements relevant to the software. Considering that such model is correct and its interpretation is well defined, one can use it to automatically generate functional test cases [21]. This technique, in which test cases are automatically derived from a model, is called Model Based Test[31].

Since models are commonly based on finite state machines, the test cases in Model Based Test are often paths in the model. A path corresponds to a sequence of consecutive transitions. There are several ways to explore the model to obtain the paths. Depending on the complexity of the model and the exploration mode chosen, the number of test cases found will be huge. In fact, if one searches for all possible paths, the number of test cases will be infinite if the model contains cycles.

Hence, there are several criteria intended to guide the model exploration and generate the paths as test cases, the so called Requirement Based Test. Some of the them are described below and with further details in [31]:

- **All transitions**

  A criterion that can be used to obtain test cases from statechart specifications. It requires that every transition should be exercised at least once during testing.

- **All simple paths**

  Another criterion that can be used with statecharts. Since all paths is an impossible achievement given the possibility that an infinite number of paths may exist, it is possible to restrict it such that every simple path in the model is exercised at least once during testing.

  It is important to note that even though this and the previous criterion seem similar to the ones described in 2.2.3, they are distinct. The ones described in this section are based on functional requirements and therefore constitute a kind of black-box test. The ones described in 2.2.3 are based on the code implementation and are a type of white-box test.

- **Distinguishing Sequence**

  This criterion should be used for finite state machine models. Besides that, the finite state machine must have the restrictions of being deterministic, complete, strongly connected and minimal.

  First, a distinguishing sequence, SD, is searched. SD is an input sequence such that when applied to each state in the machine, the output produced is different, making it possible to identify the initial state to which SD was applied. The distinguishing sequence may not exist, in this case the criterion cannot be applied.

  Second, for each transition t, an input sequence from the initial state up to and including t is generated. That sequence is called a $\beta - sequence$.

$\beta - sequence$	Transition	$\beta - sequence$ + SD
A	(0, 3)	A B B
B	(0, 0)	B B B
A A A A A	(1, 4)	A A A A A B B
A A A A B	(1, 2)	A A A A B B B
A A A A	(2, 1)	A A A A B B
A A A B	(2, 3)	A A A B B B
A A	(3, 4)	A A B B
A B	(3, 3)	A B B B
A A A	(4, 2)	A A A B B
A A B	(4, 0)	A A B B B

Table 2.1: *Distinguishing Sequence cases for automaton 2.6. SD = "B B".[31]*

We can then concatenate SD to the end of each $\beta - sequence$ to obtain the test cases. When one of these test cases is executed, it will be specifically testing a transition and checking if this transition reached the expected state.

For the automaton in figure 2.6, we have one possible SD = **B B**. In Table 2.1 we have the complete set of $\beta - sequences$ to demonstrate the Distinguishing Sequence criterion.

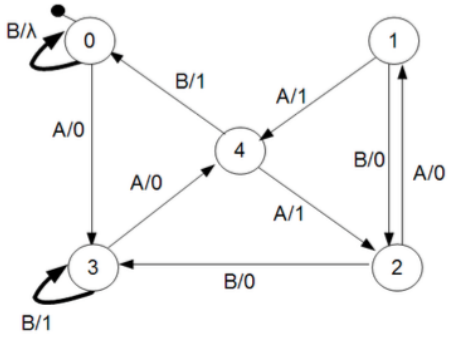


Figure 2.6: *Automaton extracted from [31]*

Let us take transition $(1, 4)$ in Figure 2.6 as an example. Suppose we already performed the first phase of the criterion and discovered that $SD = "BB"$ for the given automaton. To test $(1, 4)$ we still need to create the corresponding $\beta - sequence$. We explore the automaton from the initial state up to state 1 and store the labels of the covered transitions. For transition $(1, 4)$, this exploration results in the path $0 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1$ with labels $AAAA$. Once we get to state 1, we proceed to state 4 with input A. We concatenate this last input label to the end of the labels discovered during exploration, resulting in the $\beta - sequence$ $AAAAA$. Finally, to conclude the test case creation, we concate the value of SD to the end of the $\beta - sequence$: $AAAAABB$. The analogous is applied to every other transition in the automaton.

- **Unique Input-Output**

  As mentioned in the previous criterion, the machine might not have a distinguishing sequence. Even in this case, it is still possible to identify each state based not only on the input, but on the output as well.

State	UES
0	B/λ
1	A/1 A/1
2	B/0
3	B/1 B/1
4	A/1 A/0

Table 2.2: *UES sequences for automaton 2.6.[31]*

$\beta - sequence$	Transition	$\beta - sequence$ + UES
A	(0, 3)	A B B
B	(0, 0)	B B
A A A A A	(1, 4)	A A A A A A A
A A A A B	(1, 2)	A A A A B B
A A A A	(2, 1)	A A A A A A
A A A B	(2, 3)	A A A B B B
A A	(3, 4)	A A A A
A B	(3, 3)	A B B B
A A A	(4, 2)	A A A B
A A B	(4, 0)	A A B B

Table 2.3: *Unique Input-Output cases for automaton 2.6.[31]*

This criterion can also be used for finite state machine models that are deterministic, complete, strongly connected and minimal.

First, for each state, a unique input-output sequence, UES, is searched. In each state, a breadth search is done and at every step, it is checked if the input and output is unique in comparison to the other states.

Second, a process to find $\beta - sequences$ is performed in the same way as in the previous criterion.

Then, we can check to which state each final transition of the $\beta - sequences$ leads to by applying the UES sequences and observing their output.

For the automaton in Figure 2.6, Table 2.2 shows the possible UES for each state. In Table 2.3 the whole set of $\beta - sequences$ and their concatenation with the respective UES are shown. Both tables together illustrate the Unique Input-Output criterion.

We present in Chapter 3 a method to automatically generate test cases for well defined statechart specifications.

2.4 Model checking

Formal methods refers to the use of precise logical and mathematical methods to reason about properties of the system [10]. They contribute to the software and hardware engineering fields with formal specification and formal verification techniques, for example, aiming to assure the reliability of the system or hardware. Formal verification techniques, such as

model checking, have the goal to verify the correctness, or absence of faults, of some given program code or design against its formal specifications[29].

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model[3]. Typically, there is a hardware or software system specification containing the requirements, and we wish to verify that certain properties, such as nonexistence of deadlocks are valid for the model of the system. The specification is the basis for what the system should and should not do, therefore it is the source for the process of creating properties.

Testing and model checking have the common target of finding bugs. But as stated by Dijkstra[7], program testing can at best show the presence of errors but never their absence. On the other hand, a model checker will find a counter example if there is a violation of a given specification. It is a rigorous method that exhaustively explores all possible behaviours of the system under consideration [26]. This is the main feature that distinguishes testing and model checking: the last one can prove the absence of bugs[14].

According to [3], the model checking process can be divided into three phases:

1 **Modelling phase**

Model the system under consideration using the model description language of the model checker and specify the properties to be verified using the property specification language.

2 **Running**

Run the model checker to check the validity of the properties in the system model.

3 **Analysis**

In case a property was violated, then one should analyse the counter example generated by the model checker. It might be the case that the property does not truly reflect the requirement and it needs to be specified correctly. One other possible conclusion, if the property was defined appropriately, is that the model actually contains an error and it needs improvement. Then, verification has to be restarted with the improved model. But, in case the model contains no errors and it represents the system design, the violation of a property indicates that the design is incorrect and needs to be fixed. The verification, then, will be redone with a new model based on an improved design.

Otherwise, if no violation was detected, the model is concluded to posses all desired properties.

We will focus on the definition of properties in the modelling phase, since property specification is one of the goals of this project.

2.4.1 Property definition

The formal properties to be validated are mostly obtained from the system's specification[3]. Hence, a property specifies a certain behaviour of the system that is being considered. One has to manually read through the specification and manually define relevant properties to the system. Identifying which scenarios and behaviours should be considered when specifying properties tends to be a difficult creative process, in which human intervention becomes necessary.

The person responsible for writing the formal property specification must have a mathematical background and knowledge of the specification language [13]. Many model checkers

such as SPIN and NuSMV accept as input properties written in Linear Temporal Logic (LTL), which is difficult to write, read and validate. Therefore, translating system requirements to formal properties is not an easy task.

One classic example for property definition is the absence of deadlocks, but properties can also specify safety protocols[23], occurrence and order of events. Let's consider, for example, that one of the requirements of a system is that always after a call to the method *open*, to open a file, there should be a call to the method *close*, to close the file. One could specify this LTL property to verify the requirement as:

$$\Box(open \rightarrow \Diamond close)$$

The LTL syntax can be found in Appendix A. Using LTL, one can specify many other properties that express predicates over time.

2.4.2 Specification patterns

The effectiveness of the assurance offered by model checking depends on the quality of the formal properties that were defined [13]. To assist this process, a set of property specification patterns were proposed in [8]. A property specification pattern is a high-level property template that can be adapted based on the requirements to be verified. The patterns were obtained from a survey of 555 specifications collected from 35 different sources, including literature and several industrial projects [9].

A hierarchy was established to facilitate browsing through the patterns and picking the most appropriate one to one's need. The higher level of the hierarchy is split into the occurrence and order categories described in more details in the next subsections.

For each pattern, a scope is required to define an interval of the specification in which the property should hold. The available scopes are: global, before Q, after Q, between Q and R, after Q until R, where Q and R are states or events in the specification.

Occurrence patterns

Occurrence patterns establish the occurrence or absence of a determined event or state in a certain scope of the specification. They can be further classified into:

- Absence: a given state or event does not occur within the scope.
- Existence: a given state or event must occur within the scope
- Bounded existence: a given state or event must occur k times within the scope
- Universality: a given state or event occurs throughout the scope

Order patterns

Order patterns provide descriptions for the order in which events or states occur. They are divided in categories:

- Precedence: a state or event P should always be preceded by a state or event Q within the scope
- Response: a state or event P should always be followed by a state or event Q within the scope

- Chain precedence: a sequence of states or events $P_1, ..., P_n$ should always be preceded by a sequence of states or events $Q_1, ..., Q_n$
- Chain response: a sequence of states or events $P_1, ..., P_n$ should always be followed by a sequence of states or events $Q_1, ..., Q_n$

2.5 Sequential pattern mining

Sequential pattern mining is a topic in data mining that discovers frequent subsequences as patterns in a sequence database [19]. Each sequence in a sequence database is called data-sequence and contains typically an ID and transactions ordered generally by time, where each transaction is a set of items.

The problem is to find all sequential patterns with a user specified minimum support, where the support of a sequential pattern is the percentage of data-sequences that contain the pattern [28]. In other words, if a user inputs a percentage p and a sequence database D is given, then the mining will return the set of patterns that appear in at least $p\%$ of the data-sequences. The formal definition can be found in [19] and in [27].

There are several applications for the problem, such as analysis of customer behaviour, purchase patterns in a store and study of DNA sequences. In section 2.5.1, a practical example is described based on [19].

Notation

A data-sequence S that has ID T and $n \geq 1$ ordered transactions $t_1, t_2, ..., t_n$ is denoted by $S = [T < t_1, t_2, ..., t_n >]$.

Each transaction t_i that is a set of $m \geq 1$ items $l_{i_1}, ..., l_{i_m}$ and is denoted by $t_i = (l_{i_1}, ..., l_{i_m})$. Thus, $S = [T < (l_{1_1}, ..., l_{1_m}), ..., (l_{n_1}, ..., l_{n_m}) >]$.

To simplify the notation in the case in which each transaction contains only one item, we can avoid the parenthesis, for example: if $t_i = (l_i)$ for all $1 \leq i \leq n$ than it is possible to write $S = [T < l_1 l_2 ... l_n >]$.

2.5.1 An example: Web usage mining

Web usage mining, also known as web log mining, is an important application of sequential pattern mining. It is concerned with finding frequent patterns related to user navigation from the information presented in web system's log. Considering that a user is able to access only one page at a time, the data-sequences would only have transitions with a single event each.

In an e-commerce application, for instance, we can have the set of items $I = a, b, c, d, e, f$ representing products that can be purchased. The occurrence of one of these items in a transaction means that a user accessed the page of such item.

Suppose the sequence database contains the following data-sequences extracted from the log: $[T1 < abdac >], [T2 < eaebcac >], [T3 < babfaec >]$ and $[T4 < abfac >]$. In this case, the analysis of the first transaction allows us to conclude that user $T1$ accessed the pages of products a, b, d, a and c in this order. By applying the web usage mining technique with support of 90%, a manager would notice that $abac$ is a frequent pattern, indicating that 90% of the users who visited product a then visit b, then return to a and later visit c. Hence, an offer could be placed in product a, which is visited many times in sequence, to increase the sales of other products.

2.5.2 Sequential pattern mining algorithms

There are several algorithms to perform the sequential pattern mining, but they differ in two aspects[19]:

- The way in which candidate sequences are generated and stored. The goal is to reduce the amount of candidates created as well as decrease I/O costs.
- The way in which support is counted and how candidate sequences are tested for frequency. The goal is to eliminate data structures used for support or counting purposes only.

Considering these topics, algorithms for sequential pattern mining can be divided in two categories: apriori-based and pattern-growth

Apriori-based algorithms

These algorithms mainly rely on the property that states that if a sequence s is infrequent, then any other sequence that contains s is also infrequent. The *GSP*[28] algorithm is an example in this category.

Apriori-based algorithms use the *generate-and-test* method to obtain the candidate patterns: the pattern is grown one item at a time and tested against the minimum support. By taking this approach, they have to maintain the support counting for each candidate and test it at each iteration of the algorithm.

In general, algorithms in this category generate an explosive number of candidate sequences, consuming a lot of memory. *GSP*, for instance, generates a combinatorially explosive number of candidates when mining long sequential patterns. This is the case of a DNA analysis application, in which many patterns are long chains.

In addition, since they need to check at each iteration for the support count, multiple scans of the database are performed, which requires a lot of processing time and the I/O is very costly. In general, to find a sequential pattern of length l, the apriori-based method must scan the database at least l times. For problems in which long patterns exist, this feature increases the cost of the application.

Pattern-growth algorithms

Pattern-growth algorithms try to use a certain data structure to prune candidates early in the mining process. Besides that, the search space is partitioned for efficient memory management. *PrefixSpan*[27] is an example of such an algorithm and it is used in our implementation due to its adequately to our problem.

The general idea behind *PrefixSpan* is as follows: Instead of repeatedly scanning the entire database, generating and testing large sets of candidate sequences, one can recursively project a sequence database into a set of smaller databases associated with the set of patterns mined so far and, then, mine locally frequent patterns in each projected database[27].

To reduce the projections size and the number of access, *PrefixSpan*, sorts the items inside each transaction and creates the projected databases based on patterns' prefixes. In order to do so, the algorithm assumes that the order of items in a transaction is irrelevant, and only the order of the whole transactions matters to the problem.

Differently from apriori-based algorithms, *PrefixSpan* does not generate or test candidate sequences. Patterns are grown from the shorter ones and projected databases keep shrinking. This is relevant in practice because, in general only a small set of sequential patterns grows

long in a database and, thus, the number of sequences in a projected database usually reduces when prefix grows.

In the worst case, *PrefixSpan* constructs a projected database for every sequential pattern. With the intention to reduce the number of projected databases and improve the performance of the algorithm, [27] proposes a technique called *pseudo partition* that may reduce the number and size of the projected databases.

2.6 Working example

In this section we describe the requirements of a system that are going be explored as examples in the next chapters. Consider an e-commerce portal of a telecommunication company in which users can buy new cellphones or change their current mobile plan. The requirements are further described in the next subsections.

2.6.1 Requirement 1: Change current mobile plan

A user can change their current mobile plan by adding a new one to their shopping cart. But, in order to conclude the change, the user must first log in, so the system can retrieve their line number.

The system should check whether the user is an employee of the company or not. If the user is an employee, the plan change will not be allowed, because employees have a special plan with discount. Otherwise the system should proceed in the validation flow.

It is also necessary to check whether the user is committed to a loyalty contract, in which case, he or she will not be allowed to conclude the process. If no loyalty contract was found in the records, the user is redirected to checkout and is able to finish the plan change.

In any case in which the continuation of the process was not allowed, the user's shopping cart should become empty.

2.6.2 Requirement 2: Process orders received in the portal

After a purchase of cellphones was conclude in the portal, the order must be processed and converted in a XML file. An order is made of entries and each entry contains a product (the cellphone) and the quantity that was bought.

The information in each entry should be processed and written in the XML file, which will be sent to an integration layer.

At the same time the order is being processed, a job to send an email, informing the user that the order is being processed, should be triggered.

2.6.3 Requirement 3: Process orders received by file

This e-commerce offers, to selected customers, the feature of purchasing products through a file. This is specially useful if the customer is another company that wishes to buy a great amount of different products.

Each line of the file should contain the product code and the corresponding quantity, both delimited by the character "|".

All the information in the file should be processed and inserted in a XML file that will first be sent to an integration layer and then to a management software.

2.6.4 Requirement 4: Update stock levels of products

When a product (a cellphone) is out-of-stock, users can still demonstrated their interest in buying it by leaving their email on the waiting list. Once the product is back in stock, they will be informed by email.

The admin of this e-commerce portal is allowed to execute a job to clear the reservation on products and update their respective stock levels.

The admin is also capable of executing a job to send emails to those users who are interested in products that were out-of-stock.

These two jobs should be allowed to run in parallel.

Chapter 3

Test case generation for statecharts

In this project, we implemented the test case generation for statecharts based on the criteria described in [5]. We test all transitions by visiting every state and triggering events for every transition that starts on the given sate.

In the next sections of this chapter we describe the techniques and algorithms used for automatically creating test cases for statecharts.

3.1 Test cases for simple statecharts

In this section, we consider only statecharts that do not contain hierarchy and concurrency. Statecharts with hierarchy and concurrency will be explained afterwards.

Each test case created will comprise:

- **A transition**. The transition that is being tested.
- **An origin state**. The state from which the transition being tested is leaving from.
- **A test path**. The path in the statechart used to activate the transition being tested.
- **An expected state**. The state to which the transition being tested should redirect.

We start by making sure every reachable state in the statechart is covered. In order to do so, for each state s in the statechart, we construct a path p from the initial state to s. The path p in said to be the *coverage path of* s. All coverage paths generated are stored in a set called *State Cover*, which is denoted by C. Therefore, C is a set of sequence of transition labels, such that we can find an element from this set to reach any desired state starting from the initial one [5].

Since there is no hierarchy or concurrency in the statechart considered at this point, the construction of C is similar to covering states of an automaton and it can be done through a depth search of the states. Find below a pseudocode for the method:

```
//Wrap method to contrusct the State Cover
//It receives as argument the initial state of a simple statechart
Set constructSetC(State initialState) {

  Set setC = new Set();

  Path emptyPath = "";

  List visited = new List();
```

```
  return constructSetCRec(initalState,emptyPath,setC);
}
```

Listing 3.1: *State Cover construction wrapper for simple statecharts*

```
//Recursive function that will do all the work
//returns the State Cover set, or set C
Set constructSetCRec(State s, Path p, Set setC, List visited) {

  visited.add(s);

  setC.add(s,p);

  for (Transition t in s.getOutgGoingTransitions()) {

    State nextState = t.getDestiny();

    if (!visited.contains(nextState)) {

      Path nextCoveragePath = p + t.getLabel());

      constructSetCRec(nextState,nextCoveragePath,setC,visited);
    }
  }

  return setC;
}
```

Listing 3.2: *Recursive State Cover construction for simple statecharts*

Now that we have the coverage for every reachable state, we need to trigger each transition on each state and create the test cases. For each transition there will be a test case, thus every transition in the diagram will be exercised at least once during testing.

Consider transition $t = (s, l, q)$, where s is the original state, l is the event label that triggers t and q is the incoming state. In the aforementioned algorithm, we computed that s has coverage path p such that $p \in C$ and p is a sequence of labels. The test case TC for transition t ($t = (s, l, q)$) concatenates event labelled by l to the end of p that reaches state q. The process is repeated to all transitions in the statechart. The following algorithm describes the test cases creation:

,

```
//Function that prints the test cases for all transitions in a statechart
void generateTestCases(Statechart sc, Set setC) {

  for (State s in sc.getStates()) {

    print("At state "+s.getName());

    Path coveragePath = setC.getCoveragePath(s);

    for (Transition t in s.getOutgGoingTransitions()) {

      print("Test case for "+t.getLabel());

      Path testPath = coveragePath + t.getLabel();

      print("Test path: "+testPath);
```

```
        State expectedState = t.getDestiny();

        print("Expected state: "+expectedState);
      }
    }
  }
```

Consider the statechart in Figure 3.1, a model for purchase flow of a telco e-commerce. The model describes the flow for users changing their cellphone plan. They will be able to change plan if they are not employees from the telco company and are not committed to a loyalty contract. Additionally, they must perform the login so that the system is able to retrieve their information. This statechart model represents the requirement described in Subsection 2.6.1.

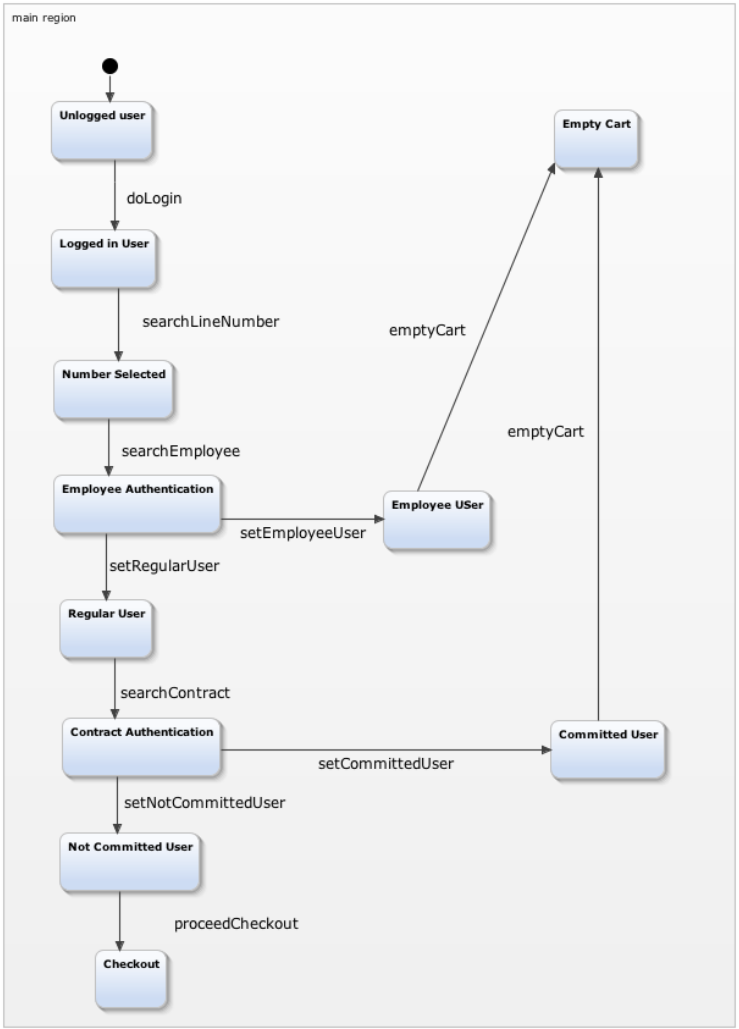


Figure 3.1: *A statechart for the plan change in a telco e-commerce*

Since, hierarchy and orthogonality are not used in the presented statechart (Figure 3.1), the technique presented above can be applied straight forward.

The construction of the *State Cover* set, or C, is performed as follows:

1. We start at the initial state *Unlogged User*. Since it is the initial state, its coverage path is the empty string, denoted by e.

2. Next, we recursively visit the states that can be reached from *Unlogged User*. In our example, we get to state *Logged in User*. In order to get to this state, it was necessary to go through transition *doLogin*. Therefore, the coverage path for *Logged in User* is *e doLogin*. Note that we concatenated the coverage path of the previous state to the undertaken transition.

Once the construction of C is concluded, we have the following coverage paths:

State	Coverage path
Unlogged user	e
Logged in User	e doLogin
Number Selected	e doLogin searchLineNumber
Employee Authentication	e doLogin searchLineNumber searchEmployee
Employee User	e doLogin searchLineNumber searchEmployee setEmployeeUser
Empty Cart	e doLogin searchLineNumber searchEmployee setEmployeeUser emptyCart
Regular User	doLogin searchLineNumber searchEmployee setRegularUser
Contract Authentication	doLogin searchLineNumber searchEmployee setRegularUser searchContract
Committed User	doLogin searchLineNumber searchEmployee setRegularUser searchContract setCommittedUser
Not Committed User	doLogin searchLineNumber searchEmployee setRegularUser searchContract setNotCommittedUser
Checkout	doLogin searchLineNumber searchEmployee setRegularUser searchContract setNotCommittedUser proceedCheckout

Note that the empty string e appears in the paths due to the unlabelled transition leaving the initial state.

Then, we have to create a test case for every transition leaving each state. Let's take state *Employee Authentication*, for example. It has two leaving transitions: $setEmployeeUser$ and $setRegularUser$. To guarantee they are exercised at least once during testing and that they go to their appropriate states, the following test cases are needed:

- Test case for ***setEmployeeUser***

  Origin state: *Employee Authentication*

  Test Path: *e doLogin searchLineNumber searchEmployee setEmployeeUser*

  Expected state: *Employee User*

- Test case for ***setRegularUser***

  Origin state: *Employee Authentication*

  Test Path: *e doLogin searchLineNumber searchEmployee setRegularUser*

  Expected state: *Regular User*

Note that the *test path* is the coverage path concatenated with the label of the tested transition. This procedure is similarly applied to all other transitions in the model.

3.2 Test cases for complex statecharts: hierarchy

In this section, the generation of test cases for statecharts that comprise the hierarchy feature is described (a state may contain many substates and so on). We do not limit the level of nested hierarchy for the automatic generation. Consider states a and b, such that a contains b. We will say that a is the superstate of b and b is the substate of a.

One way to deal with hierarchy is eliminating it from the model by flattening the statechart, as shown in 2.1.3. If so, the statechart becomes an automaton and the techniques for simple statecharts explained in the previous section can be used. Instead, the approach taken in the present project is keeping the structure of the statechart and creating the test cases incrementally, following the technique described in [5].

Similarly to the previous case, we need to cover all states by constructing the set C and then test all transitions in the model. The construction of C, however, needs to take into account states and all their corresponding substates in order to provide the whole coverage. It is important to note that we considered only statecharts that do not have transitions between different hierarchy levels.

Given a state, we first check if it contains substates. If it does, we can compute substates' coverage paths going deeper in the hierarchy level. Later, we concatenate the coverage path of the superstate to each coverage path of the substates. Then, the coverage path of the superstate should be removed from C and the paths of the substates will be kept in C instead. If the coverage path p of a certain state s passes through a superstate q, we need to mark in p that it passed by q and that the coverage paths of q's substates should be used when creating test cases for transitions leaving s. To mark that, we will use the notation Δ_q

The algorithm to construct C is changed accordingly to comprise these new features:

```
//Recursive function that will do all the work
//returns the State Cover set, or set C
Set constructSetCRec(State s, Path p, Set setC, List visited) {

  visited.add(s);

  setC.add(s,p);

  if (s.containsSubstates()) {

    Set subSetC = constructSetC(s.getInitialSubstate());

    s.addSubpaths(subSetC);

    setC.remove(s,p);

    for (State substate in s.getSubstates()) {

      Path partialPath = subSetC.getPath(substate);

      Path substateCoveragePath = p + partialPath;

      setC.add(substate,substateCoveragePath)
    }

  }

  for (Transition t in s.getOutgGoingTransitions()) {

    State nextState = t.getDestiny();
```

```
    if (!visited.contains(nextState)) {

      if (s.containsSubstates()) {

        Path nextCoveragePath = p + Δ_s + t.getLabel());

      } else {

        Path nextCoveragePath = p + t.getLabel());

      }

      constructSetCRec(nextState,nextCoveragePath,setC,visited);
    }
  }

  return setC;
}
```

Listing 3.3: *Recursive State Cover construction for statecharts with hierarchy*

Once the set C is complete, we can create the test cases based on every transition that leaves each state. In states with no substates and whose coverage did not pass by any superstate, the previous process is applied. On the other hand, if the state's coverage path p went through a superstate, we need to expand the path with the coverage paths of the substates. In other words, for each substate with coverage path u, there will be a p' with u replacing the notation Δ_s, where s is the superstate. The algorithm for the expansion is:

```
//Pseudocode for an expansion function
//Receives the original, not expanded, path and the super state it passes
    through
//Returns the set of paths resulting from the expansion
Set pathExpansion(Path originalPath, State super) {

  Set subPaths = super.getSubpaths();

  Set expansionResults = new Set();

  for (State substate in super.getSubstates()) {

    Path subPath = subPaths.getPath(substate);

    Path expanded = originalPath.replace(Δ_super, subPath);

    expansionResults.add(expanded);
  }

  return expansionResults;
}
```

Listing 3.4: *Expansion psseudocode for a path that passes through a superstate*

If a state contains substates, however, we must transfer the origin of every transition that leaves it to each one of its substates. Consider the case that a state s has a transition $t = (s, l, q)$ and contains substates s_1, s_2 and s_3. When creating the test case for t, we must actually consider three new transitions: $t_1 = (s_1, l, q), t_2 = (s_2, l, q)$ and $t_3 = (s_3, l, q)$. The pseudocode to transfer transitions from the superstate to the substates is presented below:

```
//Add all transitions of a superstate in its substates
//Receives the superstate as argument
void transferFromSuperToSub(State super) {

  for (Transition t in super.getOutgGoingTransitions()) {

    for (State sub in super.getSubstates()) {

      sub.addOutGoingTransition(t);
    }
  }
}
```

Listing 3.5: *Pseudocode to transfer transitions from a superstate to its substates*

The pseudocode to create the final test cases is the same as the one presented in Section3.1.

Let's take the statechart in Figure 3.2 to illustrate the technique. It models an application that receives order files in an specific format and converts them into a well formatted xml. Each order file contains several lines, and each line contains a product and its corresponding amount. The application also has an integration layer: it receives the xml file and then send it to the management system. This statechart model corresponds to the requirement described in Subsection 2.6.3.

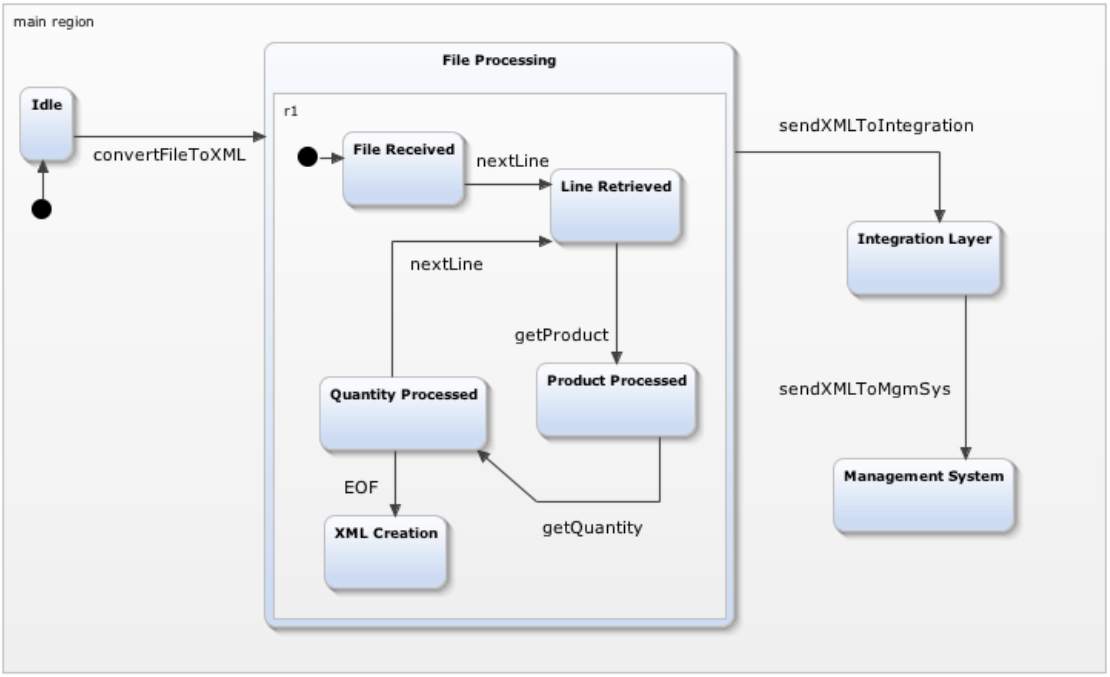


Figure 3.2: *Statechart for order file processing and transference*

Now, to construct the coverage path to *Idle* we do not need to worry about hierarchy, so approach described in the prior section (3.1) is enough.

State	Coverage path
Idle	e

Notice that once again the empty string e is present due to the unlabelled transition leaving the initial state. When creating the coverage path for state *File processing*, however,

we realise that it is a superstate. So, we go deeper in the hierarchy level to obtain the coverage paths of substates *File received, Line retrieved, Product processed, Quantity processed* and *XML creation*. Then, the following paths are added to set C:

State	Coverage path
File received	e convertFileToXML e
Line retrieved	e convertFileToXML e nextLine
Product processed	e convertFileToXML e nextLine getProduct
Quantity processed	e convertFileToXML e nextLine getProduct getQuantity
XML creation	e convertFileToXML e nextLine getProduct getQuantity EOF

Note that the coverage path for the superstate *File processing* is removed from C because we are considering its substates. Therefore, the test cases that would be created based on the superstate will be created based on the substates, instead. Also, the empty string e is added twice in each of these paths. This is necessary because they pass through two initial states with unlabelled transitions: the first one is the general initial state, and the second one is the initial state inside *File processing*.

Now, states *Integration layer* and *Management system* must also be covered. Observe that to cover these states, we need to pass by *File processing*, a superstate. Therefore, in their coverage path, we need to use an specific notation to guide the later expansion with the substates coverage paths. Notation $\Delta_{File\ processing}$ is used for this particular purpose. When creating the test cases, we need to expand the path considering the coverage of *File processing*'s substates. Thus, the coverage paths for the *Integration layer* and *Management system* states are:

State	Coverage path
Integration layer	e convertFileToXML $\Delta_{File\ processing}$ sendXMLToIntegration
Management system	e convertFileToXML $\Delta_{File\ processing}$ sendXMLToIntegration sendXMLToMgmSys

At this point, when the set C is complete, we can actually create the test cases. To illustrate this, we will examine transitions *getProduct, sendXMLToIntegration* and *sendXMLToMgmSys* more closely.

For *getProduct*, a transition leaving a substate, the process will be the same as the one presented in the Section 3.1. Hence, we have the following test case:

- Test case for ***getProduct***

  Origin state: *Line Retrieved*

  Test Path: *e convertFileToXML e nextLine getProduct*

  Expected state: *Product processed*

In the case of *sendXMLToIntegration*, a transition that leaves a superstate, we need to use the information regarding the substates to create the test cases. For each substate's coverage path p, there will be a test case for *sendXMLToIntegration* using p. Then, $sendXMLToIntegration$ is appended to each p in order to obtain the test paths:

- Test case #1 for ***sendXMLToIntegration***

  Origin state: *File Received*

  Test Path: *e convertFileToXML e sendXMLToIntegration*

  Expected state: *Integration layer*

- Test case #2 for ***sendXMLToIntegration***

  Origin state: *Line Retrieved*

  Test Path: *e convertFileToXML e nextLine sendXMLToIntegration*

  Expected state: *Integration layer*

- Test case #3 for ***sendXMLToIntegration***

  Origin state: *Product Processed*

  Test Path: *e convertFileToXML e nextLine getProduct sendXMLToIntegration*

  Expected state: *Integration layer*

- Test case #4 for ***sendXMLToIntegration***

  Origin state: *Quantity Processed*

  Test Path: *e convertFileToXML e nextLine getProduct getQuantity sendXMLToIntegration*

  Expected state: *Integration layer*

- Test case #5 for ***sendXMLToIntegration***

  Origin state: *XML Creation*

  Test Path: *e convertFileToXML e nextLine getProduct getQuantity EOF sendXMLToIntegration*

  Expected state: *Integration layer*

As for transition *sendXMLToMgmSys*, the expansion of *Integration layer*'s coverage path is pending. Similarly to what we did for the transition *sendXMLToIntegration*, we also need to consider the paths of *File processing*'s substates. Therefore, the test cases for *sendXMLToMgmSys* are:

- Test case #1 for ***sendXMLToMgmSys***

  Origin state: *Integration Layer*

  Test Path: *e convertFileToXML e sendXMLToIntegration sendXMLToMgmSys*

  Expected state: *Integration layer*

- Test case #2 for ***sendXMLToMgmSys***

  Origin state: *Integration Layer*

  Test Path: *e convertFileToXML e nextLine sendXMLToIntegration sendXMLToMgmSys sendXMLToMgmSys*

  Expected state: *Integration layer*

- Test case #3 for ***sendXMLToMgmSys***

  Origin state: *Integration Layer*

  Test Path: *e convertFileToXML e nextLine getProduct sendXMLToIntegration sendXMLToMgmSys*

  Expected state: *Integration layer*

- Test case #4 for ***sendXMLToMgmSys***

  Origin state: *Integration Layer*

  Test Path: *e convertFileToXML e nextLine getProduct getQuantity sendXMLToIntegration sendXMLToMgmSys*

  Expected state: *Integration layer*

- Test case #5 for ***sendXMLToMgmSys***

  Origin state: *Integration Layer*

  Test Path: *e convertFileToXML e nextLine getProduct getQuantity EOF sendXMLToIntegration sendXMLToMgmSys*

  Expected state: *Integration layer*

Notice that, in each case, $\Delta_{File\ processing}$ in the *Integration layer*'s coverage was replaced by a substate's coverage path.

3.3 Test cases for complex statecharts: orthogonality

Now, we shall consider statecharts that comprise orthogonality (states in concurrent regions).

A method to generate test cases dealing with orthogonality is to eliminate it by flattening the statechart, as explained in 2.1.3. The elimination of orthogonality would be done with the Cartesian product of all states and transitions, causing an explosion in the number of result states and transitions [5].

To avoid states and transitions explosion and still be able to cover all states and test all transitions, [5] offers an alternative approach to refine the concurrency requisites. In the present project, we chose to use the strong concurrency refinement:

- **Strong concurrency**

  This refinement allows us to test concurrent components separately. Transitions from each concurrent region are triggered one-by-one in different steps. In this case, we consider that the concurrent regions are placed in units which either run in parallel or in different processors. So no concurrent region may cause missing transitions in another one or misdirected transitions.

The communication resources, as broadcasting, should be disabled during testing since it could cause series of transitions to occur. A chain reaction would be an example of such series and would not be expected by the test cases listed using this implementation.

In order to create the test cases for statecharts that comprise concurrent regions, we first compute the coverage paths for each concurrent region separately. Then, similarly to the case with hierarchy, we combine these paths with the coverage path of the state that contains the concurrent regions. After obtaining the coverage path for all states, set C is complete.

In the pseudocode below, we consider that substates are in a region inside the superstate. To apply to statecharts with hierarchy, discussed in section 3.2, we must consider that the superstate have only one internal region, which will contain the substates. In statecharts with concurrency, the concurrent elements must be in different regions inside a superstate (Figure 3.3 serves as an example to illustrate all this). The solution presented in the pseudocode can be applied to states with orthogonality as well as to ones with hierarchy only. It first calculates the *State Cover* set, set C:

```
//Recursive function that will do all the work
//returns the State Cover set, or set C
Set constructSetCRec(State s, Path p, Set setC, List visited) {

  visited.add(s);

  setC.add(s,p);

  if (s.containsSubRegions()) {

    for (Region r in s.getSubregions()) {

      Set subSetC = constructSetC(r.getInitialSubstate());

      r.addSubpaths(subSetC);

      setC.remove(s,p);

      for (State substate in s.getSubstates()) {

        Path partialPath = subSetC.getPath(substate);

        Path substateCoveragePath = p + partialPath;

        setC.add(substate,substateCoveragePath)
      }
    }

  }

  for (Transition t in s.getOutgGoingTransitions()) {

    State nextState = t.getDestiny();

    if (!visited.contains(nextState)) {

      if (s.containsSubstates()) {

        Path nextCoveragePath = p + Δs + t.getLabel());

      } else {

        Path nextCoveragePath = p + t.getLabel());

      }

      constructSetCRec(nextState,nextCoveragePath,setC,visited);
    }
  }

  return setC;
}
```

Listing 3.6: *Recursive State Cover construction for a statechart with orthogonality*

Once the C set is built, the test cases for each transition of the model can be generated. This process is the same as the one described in the case with hierarchy in 3.2, since concurrent states are inside a superstate.

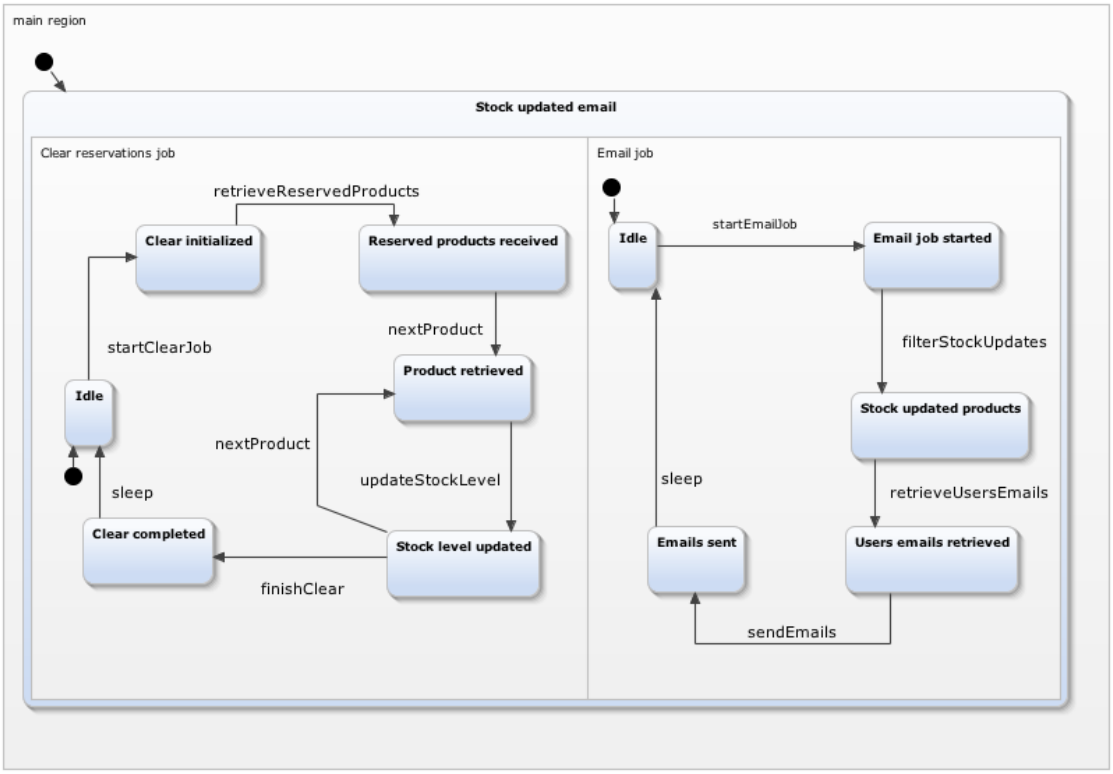


Figure 3.3: *Statechart model for concurrent jobs: clear reservations job and send email job*

To illustrate this case, we can look at the example provided in Figure 3.3. It models an e-commerce application in which the administrator is allowed to execute two jobs: one is to clear all the reservations of products (region *Clear reservations job* in the figure) and the other one is to send emails to customers letting them know certain products are back in stock (region *Email job* in the figure). Both jobs can run in parallel if the administrator wishes, thus their regions are modelled in a concurrent way in the statechart. This statechart corresponds to the requirement described in Subsection 2.6.4.

According to our refinement, the coverage paths will be created for substates in *Clear reservations job* and *Email job* separately. First, the substates in *Clear reservations job* concurrent region are inside state *Stock update email*, hence we need to apply the algorithm presented above. Note that since the coverage path of *Stock update email* is just the empty string e, it will not have a great impact on the substates paths. Let's consider *Product retrieved* and *Stock level updated* to exemplify the results:

State	Coverage path
Product retrieved	e e startClearJob retrieveReservedProducts nextProduct
Stock level updated	e e startClearJob retrieveReservedProducts nextProduct updateStockLevel

The coverage paths for substates in region *Email job* can be built using a similar process. For instance, the coverage path for *Email sent* is:

State	Coverage path
Email sent	e e startEmailJob filterStockUpdates retrieveUsersEmails sendEmails

The generation of the test cases, then, is similar to the one presented in Section 3.2. But, when there is orthogonality, we need to consider the coverage paths of substates from all concurrent regions in a state during the expansion phase. For the example in Figure 3.3, since there is no state after *Stock updated email*, no expansion of coverage paths will be needed.

For example, let's consider the test cases for transitions *nextProduct*, from state *Stock level updated*, and *sleep*, from state *Email sent*. We need to append these transition labels to the coverage path of their origin state. Therefore, *nextProduct* will be appended to the coverage path of *Stock level updated*, and *sleep* to the coverage path of *Email sent*:

- Test case for ***nextProduct***

  Origin state: *Stock level updated*

  Test Path: *e e startClearJob retrieveReservedProducts nextProduct updateStockLevel nextProduct*

  Expected state: *Product retrieved*

- Test case for ***sleep***

  Origin state: *Emails sent*

  Test Path: *e e startEmailJob filterStockUpdates retrieveUsersEmails sendEmails sleep*

  Expected state: *Idle*

Chapter 4

Formal property extraction from test cases

In this chapter we propose a technique to automatically specify formal properties from the test cases generated in the previous chapter. Since a great number of test cases might be generated, we should be able to identify the more relevant patterns among them. In order to do so, we will apply the mining concepts and $PrefixSpan$ algorithm presented in Section 2.5 to extract patterns, which will later be used to derive the properties.

In the next sections of this chapter we describe the techniques and algorithms used to automatically specify formal properties in LTL, linear temporal logic.

4.1 Test case mining

In Chapter 3, we presented a method to automatically generate test cases from statecharts specifications. Each test case presented consisted of a transition t we wished to test, a test path p to activate the transition and a state s which was the expected state t would redirect to.

The test path p is basically a sequence of consecutive events. Therefore, it is possible to analyse it with the concepts and notation shown in Section 2.5. Let's say that p is a sequence of events $e_1, e_2, ..., e_n$ (in this order). We can associate a sequence s_p to the test path p such that $s_p = [T_p < e_1 e_2 ... e_n >]$, where T_p is an arbitrary unique ID.

The set of test cases automatically generated from the statechart, can then be seen as a sequence database. Hence, we are able to apply sequential pattern mining algorithms, such as *PrefixSpan* (Section 2.5), to acquire the most frequent patterns in the set of test cases. The user defines the minimum support for the mining algorithm and then obtains the most frequent subsequences in the set of test paths.

The most frequent subsequences returned by the mining play an important rôle during testing, due to the fact that they are the ones mostly stressed in the testing criterion. If a subsequence $< abc >$ is considered a frequent pattern with minimum support of 60%, it means that events a, b and c will be executed in this order at least 60% of the time during the test activity.

Furthermore, mining these sequences also points out which event subsequences most test cases rely on. This means that a defect in any of them would block a considerable amount of test cases execution. Considering the previous example with pattern $< abc >$ and 60% of minimum support. If there is bug in the system that damages the execution of events a, b and c in this order, then it implies that at least 60% of the test case execution would be harmed as well, impacting on the system delivery to the client.

In conclusion, the advantage of using a mining technique is that, besides reducing the amount of sequences to be analysed, it provides subsequences that are more relevant to the testing process. Since these subsequence patterns are important to testing, they should also be important to the system execution as a whole.

4.1.1 The SPMF framework

In our implementation, we used the *Sequential Pattern Mining Framework* (*SPMF*)[12] to perform the test case mining. *SPMF* is an open-source data mining library written in Java, specialised in sequential mining. It was easily integrated with our Java code, even though it can be used as a standalone application.

It offers several mining algorithms implementations, not only for sequential mining, but for association rule and clustering classification. For the purpose of this project, we chose the provided *PrefixSpan* algorithm due to the empirical analysis presented in [27] demonstrating that it would be more efficient than other classic sequential pattern mining algorithms, such as the GSP algorithm.

The algorithm receives as input the sequence database and the minimum support value provided by the user. It then computes the most frequent subsequence patterns, which are internally stored and used during the creation of the formal properties. Note that our implementation does not output the discovered patterns, since this is not the final goal of the project. We use the subsequence patterns returned by *SPMF* for the property generation.

4.1.2 Test case mining example

To illustrate the mining technique, let's consider the test paths generated for the statechart in 3.1 presented in Table 4.1. The $PrefixSpan$ implementation of $SPMF$ requires an input file such that there must be one sequence per line, each event must be delimited by -1 and each line must end with -2.

Inputting sequences in Table 4.1 to $PrefixSpan$ with minimum support of 70%, we get as output the patterns presented in Table 4.2. We can then notice, for instance, that events *doLogin, searchLineNumber* and *searchEmployee* occur in this order in at least 70% of the generated test cases.

4.2 Generation of properties from most frequent test case patterns

Based on the patterns obtained through the usage of $PrefixSpan$ shown previously, we can automatically define formal properties to be verified. For this project we chose two property specification patterns, explained in Section 2.4.2, from [8]: the response and existence patterns. First, we go through the most frequent patterns based on a user minimum support and compute the formal response properties. Second, we try to find patterns that are common to all of the test cases to be able to apply the existence specification.

4.2.1 Response property specification

The creation of response properties is done using as input the patterns mined by $PrefixSpan$ in the previous step. In each sequence pattern, we obtain pairs of consecutive events and

Test paths for 3.1
doLogin -1 -2
doLogin -1 searchLineNumber -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setEmployeeUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setEmployeeUser -1 emptyCart -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setCommittedUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setCommittedUser -1 emptyCart -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setNotCommittedUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setNotCommittedUser -1 proceedCheckout -1 -2

Table 4.1: *Test paths obtained from the test cases generated for statechart in Figure 3.1*

doLogin -1
doLogin -1 searchEmployee -1
doLogin -1 searchLineNumber -1
doLogin -1 searchLineNumber -1 searchEmployee -1
searchLineNumber -1
searchLineNumber -1 searchEmployee -1
searchEmployee -1

Table 4.2: *Patterns extracted from sequences in Table 4.1 with minimum support of* 70%.

establish a property that the second event must respond to the first event of the pair. The global scope was used. The pseudocode can be found below:

```
//Method to write the specification of formal response properties
//It receives as argument a sequence pattern

Set propertySet = new Set();

Set extractResponseProperties(Sequence pattern) {

  for (i = 0; i < pattern.length - 1; i++) {
    Event P = pattern.getEvent(i);
    Event S = pattern.getEvent(i+1);

    Property responseProperty = □(P.getName() → ◊ S.getName());

    if (!propertySet.contains(responseProperty))
      propertySet.add(responseProperty);
  }
```

```
    return propertySet;
}
```

Listing 4.1: *Pseudocode to extract response properties from the mining results*

To illustrate this phase, take as example the test cases automatically generated for statechart in Figure3.1, presented in Table 4.1. With a minimum support of 70%, the most frequent patterns returned are displayed in Table 4.2. Applying each one of these patterns as input to the method *extractResponseProperties*, we obtain the following response properties:

Informal description	Formal specification
searchLineNumber responds to doLogin	$\Box(doLogin \rightarrow \Diamond searchLineNumber)$
searchEmployee responds to doLogin	$\Box(doLogin \rightarrow \Diamond searchEmployee)$
searchEmployee responds to searchLineNumber	$\Box(searchLineNumber \rightarrow \Diamond searchEmployee)$

Table 4.3: *Properties automatically extracted from patterns in 4.2.*

These properties reflect some of the requirements of the specification. The login has to be performed so that the system can access users information: the corresponding users line number or if users are employees of the company, for example. Besides that, according to the flow described by the statechart (Figure 3.1), the employee verification should be done after the line number is retrieved, as stated in the third property.

It is possible to realise that some of the generated properties contain events in common. Therefore, we can combine them in order to reduce the number of properties that should be verified by the model checker. Let's consider, for example, the first and second properties described in Table 4.3:

$$\Box(doLogin \rightarrow \Diamond searchLineNumber)$$
$$\Box(doLogin \rightarrow \Diamond searchEmployee)$$

Events *searchLineNumber* and *searchEmployee* respond to the same event *doLogin*. Hence, we synthesise both properties in a new more concise one:

$$\Box(doLogin \rightarrow \Diamond(searchLineNumber \wedge searchEmployee))$$

Then, we are left only with to properties to be passed to the model checker:

Informal description	Formal specification
searchLineNumber and searchEmployee responds to doLogin	$\Box(doLogin \rightarrow \Diamond(searchLineNumber \wedge searchEmployee))$
searchEmployee responds to searchLineNumber	$\Box(searchLineNumber \rightarrow \Diamond searchEmployee)$

Table 4.4: *Combined properties from 4.3.*

4.2.2 Existence property specification

In order to use the existence specification pattern, we must find sequence patterns that are present in the whole set of test cases. In other words, we should perform the test case mining stage with a minimum support of 100%. If any such pattern is found, we use the global scope and define an existence property, meaning that for every path taken, we will eventually find that pattern. Considering that a list of events was found as patterns with support of 100%, the following pseudocode can be used:

```
//Method to write the specification of formal existence properties
//It receives as argument a list of events that were found during mining
    with support of 100%

Set propertySet = new Set();

Set extractExistenceProperties(List commonEvents) {

  for (i = 0; i < pattern.length; i++) {
    Event P = pattern.getEvent(i);

    Property existenceProperty = ◊(P.getName());

    if (!propertySet.contains(existenceProperty))
      propertySet.add(existenceProperty);
  }

  return propertySet;
}
```

Listing 4.2: *Pseudocode to extract existence properties from the mining results*

Still considering test cases in Section 4.1 to illustrate the process, the only event returned by the mining phase with 100% of support is *doLogin*. Thus, we are able to specify one existence property that indicates that the event *doLogin* must occur, no matter the execution path is taken:

Informal description	Formal specification
doLogin must occur	$\Diamond(doLogin)$

Table 4.5: *Existence property extracted from 4.1.*

In the that case more than one event is present in all test cases, then we can combine them to create a single existence property to be verified by the model checker. Suppose events a, b and c are present in all test cases of a certain statechart model. Then, we could combine their existence properties in the following single property:

$$\Diamond(a \wedge b \wedge c)$$

4.3 Generation of properties for specific events

Due to the fact only the patterns returned by the sequential mining are considered to derive the property specifications, only the events that appear more often in the test cases are going to be selected to create properties. Rare events are discarded when the minimum support is too high. An immediate solution would be to set the minimum support as a lower

doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setCommittedUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setCommittedUser -1 emptyCart -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setNotCommittedUser -1 -2
doLogin -1 searchLineNumber -1 searchEmployee -1 setRegularUser -1 searchContract -1 setNotCommittedUser -1 proceedCheckout -1 -2

Table 4.6: *A database for event setRegularUser.*

setRegularUser
setRegularUser, searchContract
setRegularUser, searchContract, setNotCommittedUser
setRegularUser, searchContract, setNotCommittedUser,proceedCheckout
setRegularUser, searchContract, emptyCart
doLogin, searchEmployee, setRegularUser
doLogin, searchEmployee, setRegularUser, searchContract
doLogin, searchEmployee, setRegularUser, searchContract, setNotCommittedUser
doLogin, searchEmployee, setRegularUser, searchContract, setNotCommittedUser, proceedCheckout

Table 4.7: *Some sequence combinations generated by $PrefixSpan$ for setRegularUser.*

value, but that would cause too many patterns to be selected and many irrelevant properties might be specified, making the model checking process more costly in time and resources.

Taking these aspects into account, we propose a solution that gives to users the option to input specific events to which they want properties to be specified. This approach allows rare events to be handled properly and, since the user is able to check which properties were automatically generated with mining, becomes a general solution to formally specify event-based requirements.

For the specific event l defined by user, we initially select the test cases that contain l to construct our sequence database. Secondly, we run the $PrefixSpan$ provided by $SPMF$ with minimum support of 0, which will output all possible sequence combination present in the database that contains the event l. Then, we are able to construct response property specifications for l.

As an example, consider the *setRegularUser* event, which is not present in the properties of Table 4.3 generated with minimum support of 70%. Suppose a user wishes to obtain response properties with this event and inputs it in our implementation.

We, then, construct a new sequence database that has only sequences comprising the event *setRegularUser*. The next step is to run $PrefixSpan$ with minimum support of 0. With that input value, the algorithm will generate all possible combinations that can be derived from the specific database. To illustrate this, we provide a database in Table 4.6 and some of the possible combinations in Table 4.7.

Finally, for each sequence combination returned, we can specify the related response property. The pseudocode is similar to the one in Section 4.2.1:

```
//Method to write the specification of formal response properties
//It receives as argument a sequence combination and the specific event

Set propertySet = new Set();

Set extractResponseProperties(Sequence combination, Event specific) {

  for (i = 0; i < pattern.length - 1; i++) {
    Event P = pattern.getEvent(i);
    Event S = pattern.getEvent(i+1);
    if (P == specific or S == specific) {
      Property responseProperty = □(P.getName() → ◊ S.getName());

      if (!propertySet.contains(responseProperty))
        propertySet.add(responseProperty);
    }
  }
  return propertySet;
}
```

Listing 4.3: *Pseudocode to extract response properties for a specific event*

Considering the whole set of combinations for *setRegularUser*, the response properties below are automatically specified:

Informal description	Formal specification
setRegularUser responds to searchEmployee	$\Box(searchEmployee \rightarrow \Diamond setRegularUser)$
searchContract responds to setRegularUser	$\Box(setRegularUser \rightarrow \Diamond searchContract)$
setRegularUser responds to searchLineNumber	$\Box(searchLineNumber \rightarrow \Diamond setRegularUser)$
setCommittedUser responds to setRegularUser	$\Box(setRegularUser \rightarrow \Diamond setCommittedUser)$
setNotCommittedUser responds to setRegularUser	$\Box(setRegularUser \rightarrow \Diamond setNotCommittedUser)$
setRegularUser responds to doLogin	$\Box(doLogin \rightarrow \Diamond setRegularUser)$
emptyCart responds to setRegularUser	$\Box(setRegularUser \rightarrow \Diamond emptyCart)$
proceedCheckout responds to setRegularUser	$\Box(setRegularUser \rightarrow \Diamond proceedCheckout)$

Table 4.8: *Response properties for event setRegularUser.*

Moreover, we can summarise the extracted properties by combining them in a similar process as presented in Section 4.2.1. There are properties in which some event responds to *setRegularUser*, while in other properties *setRegularUser* responds to some event. Hence, we are left with the following two more concise properties to use in the model checking verification:

- *Informal description*: emptyCart, proceedCheckout, setCommittedUser, setNotCommittedUser and searchContract respond to setRegularUser

Formal specification: $\Box(setRegularUser \rightarrow \Diamond(emptyCart \wedge proceedCheckout \wedge setCommittedUser \wedge setNotCommittedUser \wedge searchContract))$

- *Informal description*: setRegularUser responds to doLogin, searchLineNumber and searchEmployee

  Formal specification: $\Box((doLogin \vee searchLineNumber \vee searchEmployee) \rightarrow \Diamond(setRegularUser))$

Chapter 5

Tools demonstration

To illustrate the whole process of generating test cases, extracting formal properties and how our implemented tools work, we will use as an example the statechart model given in Figure 5.1, created with the Yakindu Statechart Tools [1]. It models the requirement described in 2.6.2: The order processing of an e-commerce portal after the user concluded the purchase. An order is made of entries, each one containing a product and its corresponding quantity. The order information must be converted in a XML file that will be sent to an integration layer. In parallel to the XML preparation, a job is executed to send to customers an email informing that their order is being processed.

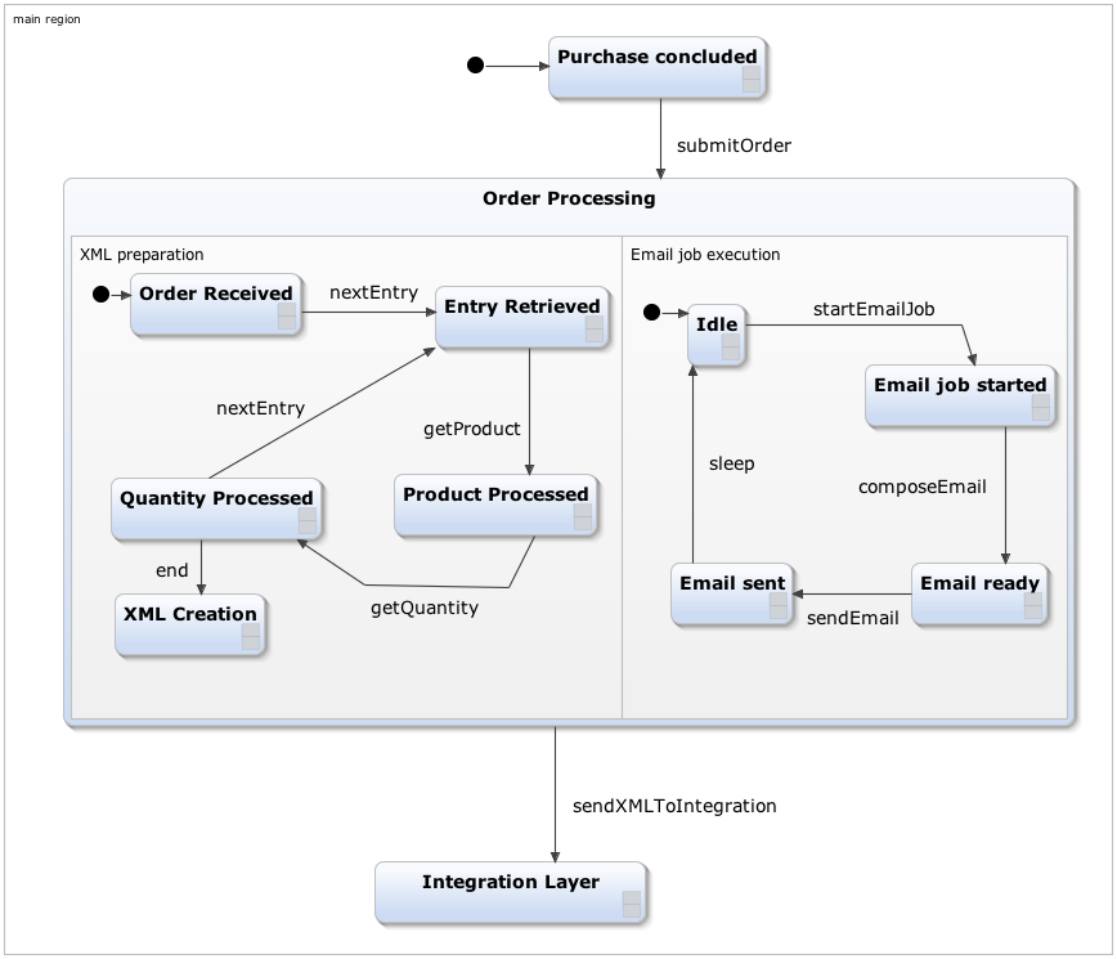


Figure 5.1: *Statechart model for order processing*

5.1 Test case generation tool

The interface of test generation tool is displayed in Figure 5.2.

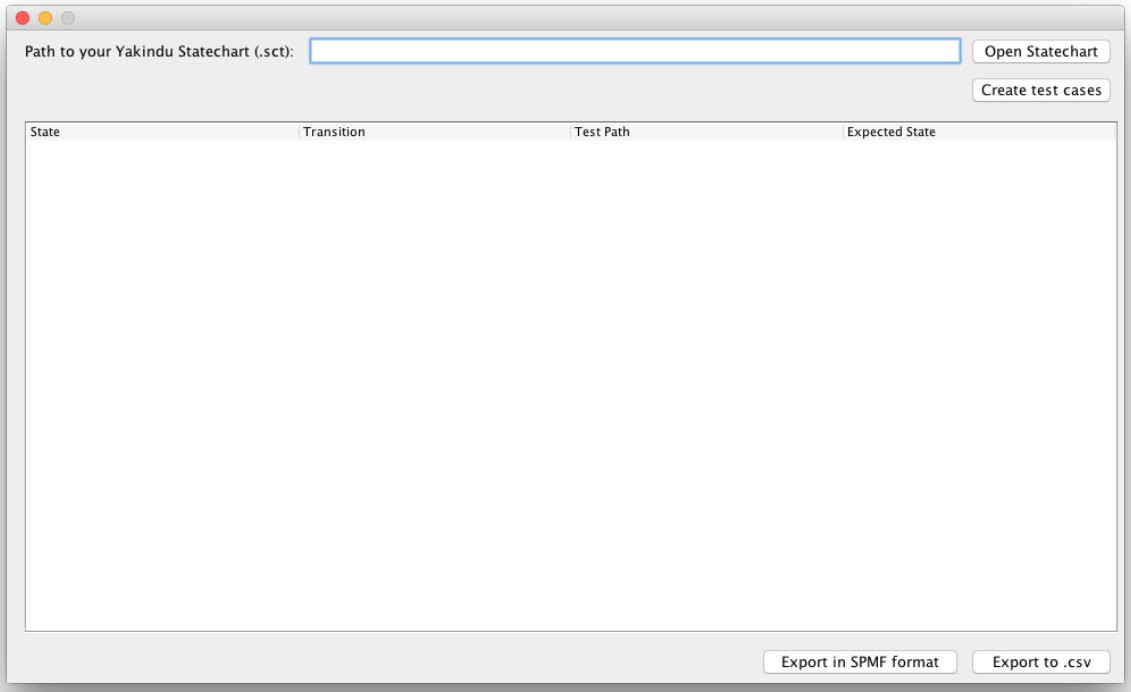


Figure 5.2: *Test case generator interface.*

We start by opening the statechart we created. Then, we click on button *Create test cases* and the created test cases will be outputted on the screen, as shown in Figure 5.3. Note that the table is splitted in four columns:

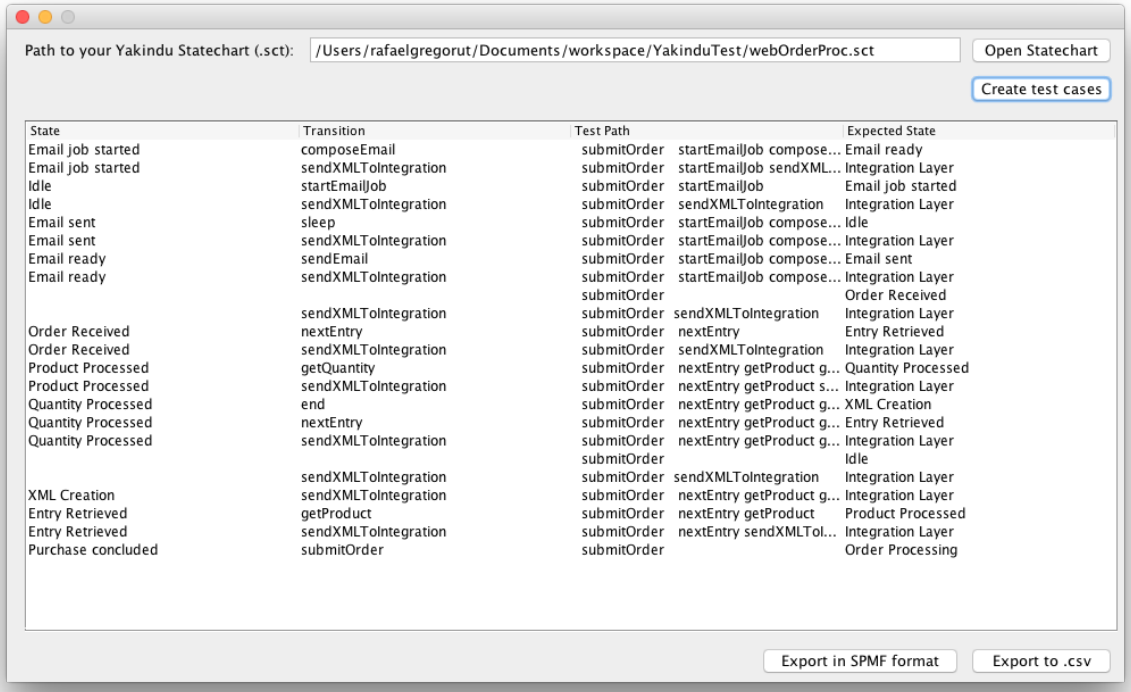
- *State*, which tells from which state we are taking a transition
- *Transition*, which informs the transition that is being tested
- *Test path*, which contains the path to execute that transition
- *Expected state*, which is the expected state we should get when that transition is taken

If we click on *export in SPMF format*, we are able to save the test paths in a text file (using a standard format for the framework *SPMF*). For our working example, the test paths saved in the standard format is displayed in Figure 5.4.

We are also able to export the information from all columns to a *.csv* file, which can be read by most spreadsheet softwares in the market. Just click on button *Export to .csv*.

5.2 Formal property specification tool

The interface of our formal property specification tool is displayed in Figure 5.5. It contains two tabs:



State	Transition	Test Path		Expected State
Email job started	composeEmail	submitOrder	startEmailJob compose...	Email ready
Email job started	sendXMLToIntegration	submitOrder	startEmailJob sendXML...	Integration Layer
Idle	startEmailJob	submitOrder	startEmailJob	Email job started
Idle	sendXMLToIntegration	submitOrder	sendXMLToIntegration	Integration Layer
Email sent	sleep	submitOrder	startEmailJob compose...	Idle
Email sent	sendXMLToIntegration	submitOrder	startEmailJob compose...	Integration Layer
Email ready	sendEmail	submitOrder	startEmailJob compose...	Email sent
Email ready	sendXMLToIntegration	submitOrder	startEmailJob compose...	Integration Layer
		submitOrder		Order Received
	sendXMLToIntegration	submitOrder	sendXMLToIntegration	Integration Layer
Order Received	nextEntry	submitOrder	nextEntry	Entry Retrieved
Order Received	sendXMLToIntegration	submitOrder	sendXMLToIntegration	Integration Layer
Product Processed	getQuantity	submitOrder	nextEntry getProduct g...	Quantity Processed
Product Processed	sendXMLToIntegration	submitOrder	nextEntry getProduct s...	Integration Layer
Quantity Processed	end	submitOrder	nextEntry getProduct g...	XML Creation
Quantity Processed	nextEntry	submitOrder	nextEntry getProduct g...	Entry Retrieved
Quantity Processed	sendXMLToIntegration	submitOrder	nextEntry getProduct g...	Integration Layer
		submitOrder		Idle
	sendXMLToIntegration	submitOrder	sendXMLToIntegration	Integration Layer
XML Creation	sendXMLToIntegration	submitOrder	nextEntry getProduct g...	Integration Layer
Entry Retrieved	getProduct	submitOrder	nextEntry getProduct	Product Processed
Entry Retrieved	sendXMLToIntegration	submitOrder	nextEntry sendXMLTol...	Integration Layer
Purchase concluded	submitOrder	submitOrder		Order Processing

Figure 5.3: *Test cases created for statechart 5.1.*

- *Properties based on test case mining*, where properties will be specified based on the results returned by sequential mining
- *Properties for specific events*, where the user is able to specify an event and properties related to it will be displayed.

Initially, we open the file created by our test case generator in $SPMF$ input format, pass a minimum support of 0.3 (30%) and click on *Specify properties* to generate the properties, as displayed in Figure 5.6

Response properties are displayed in the first panel, together with the combined properties. Existence properties are shown in the second panel together with their combinations. In both panels there are two columns:

- *Informal description*, that contains a description of the property in natural language
- *Formal property*, where the formal specification in LTL is displayed.

Notice that, in case a sequence is ignored in the mining process, its ID will be displayed in the bottom area *Not used sequences*. The ID of a sequence corresponds to the line number where the sequence is located.

If we change to tab *Properties for specific events*, we can obtain response properties for a specific event. In our example, we passed as input the event *sendEmail* and, once *Specify properties* is clicked on, the response properties are displayed as shown in Figure 5.7. Note that combined properties are also displayed.

Figure 5.4: *Test paths from 5.1 in the SPMF input format.*

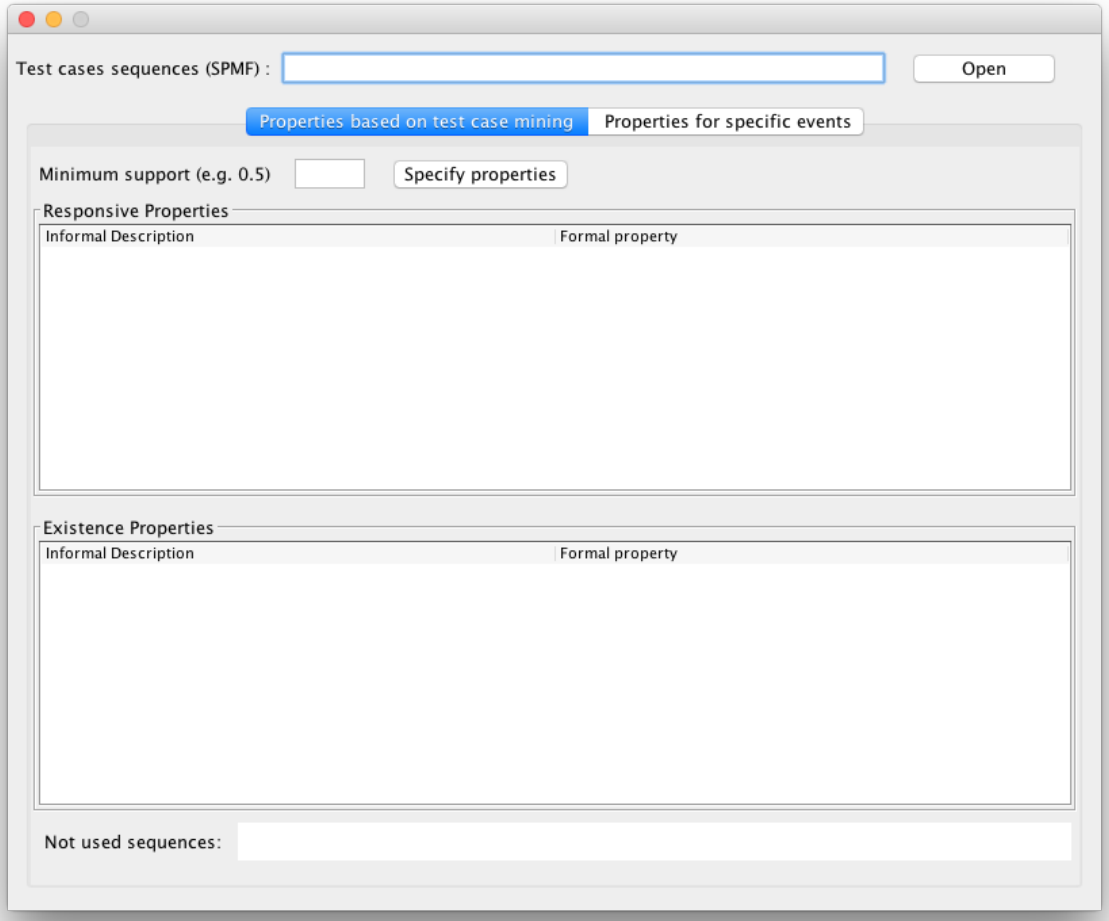


Figure 5.5: *Property generator tool interface.*

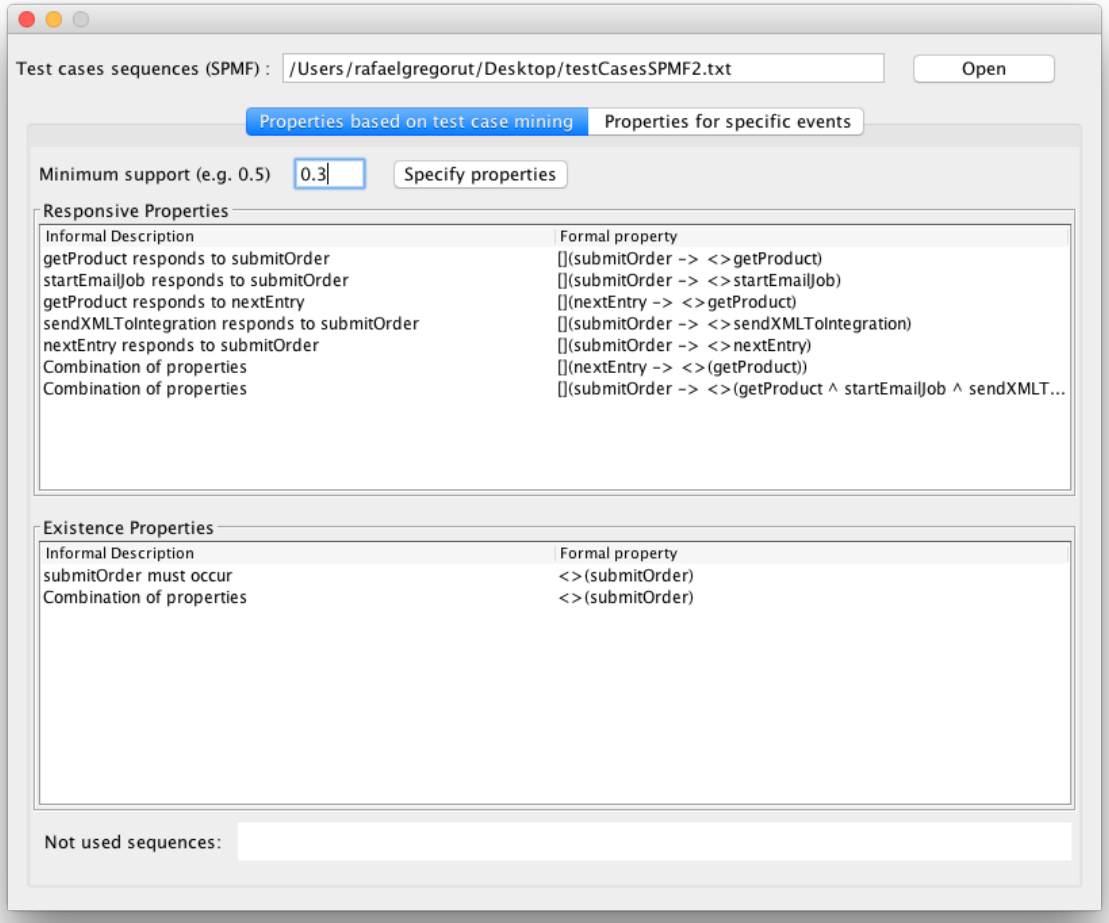


Figure 5.6: *Properties automatically specified for 5.1 based on the mining of 5.4.*

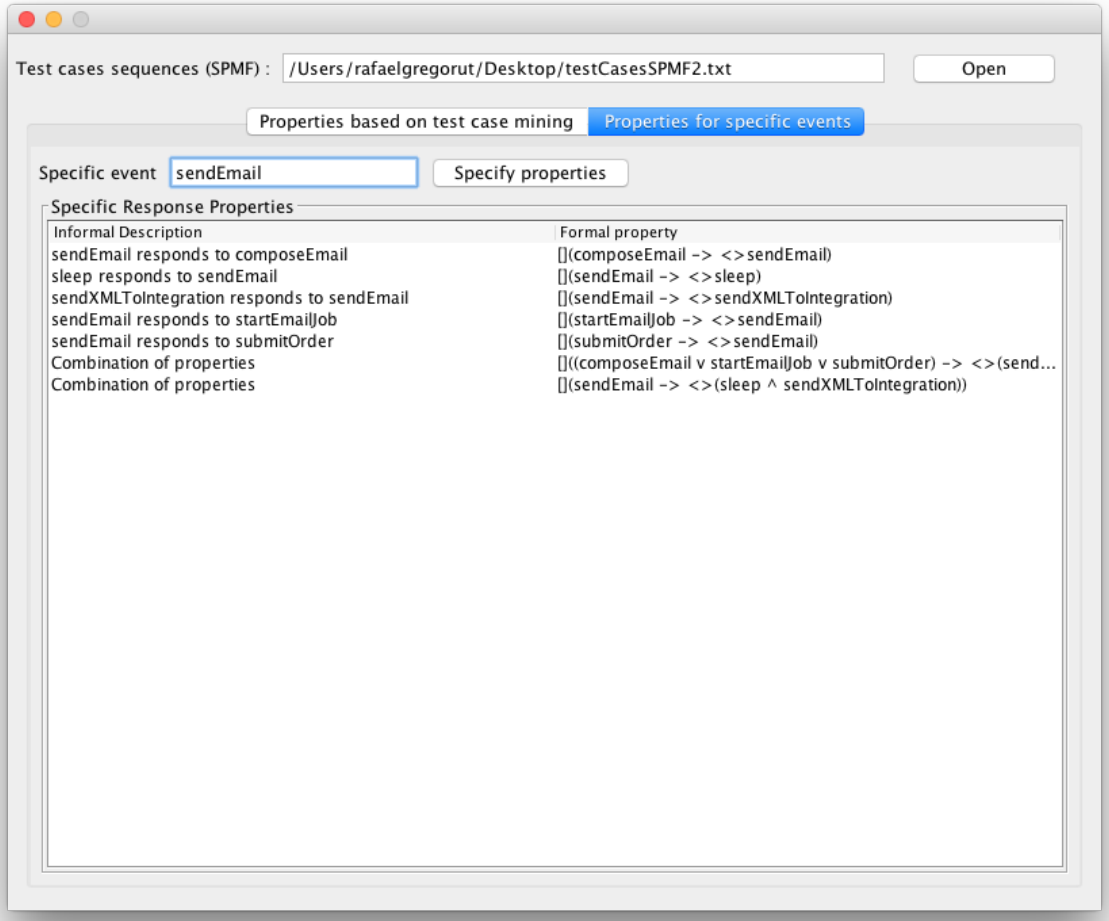


Figure 5.7: *Properties specified for sendEmail event.*

Chapter 6

Conclusion

The system specification document is a reference to be consulted throughout the entire software development cycle. For the code implementation, the specification's relevance is due to the fact that it contains client's requirements and project decisions to guide programming. In the requirements based testing, the specified requirements should also be used in test cases to validate the observed behaviour and the output of the system comparing to what was expected. However, the main problem a specification can suffer from is the lack of precision and completeness. This may cause gaps in understanding during the whole process and consequently damaging the development and testing activities.

Therefore, manually created test cases based on informal specifications tend to lack accuracy and the imprecise information they contain may lead to incorrect conclusions regarding the quality of the system under consideration. An implementation that passes all test cases successfully is not guaranteed to have no errors, since the testing may not have covered all usage scenarios and it certainly did not test all possible inputs.

Formal methods can bring formalism and contribute to the software development process. Already used in critical systems, they provide techniques with rigorous mechanisms to assure the quality and safety of the product being delivered. Even though the complexity of general systems is huge, formal approaches can be applied in different parts of the system, at different phases of the development.

Consider statecharts for example, a type of formal specification based on finite state machines. They provide ways to specify flows and scenarios with formalism and precision, even for concurrent systems. Besides that, they have the visual appeal that facilitates their comprehension. Tools, such as Yakindu, to create and even simulate their execution are available and collaborate to their spread in the software engineering community.

Formal verification can also be used in specific components of the system, such as security protocols. It does not depends on any specific input and, hence, can discover incorrect behaviours that tests would not be able to find. Moreover, formal verification can prove the absence of errors, an achievement that is not reachable by testing. Nonetheless, this approach faces practical obstacles, such as the difficulty to convert informal requisites into formal properties and the specification language that must be used.

With these issues in mind, we believe that formal techniques should be used along with testing to improve the quality of systems. The technique to generate test cases from formal statechart specifications can be used to create test cases with more precision and guide the test execution. In addition, the automatic synthesis of formal properties provide more automation to the formal verification process and assist developers to specify relevant properties of the system. The present work has given a particular solution to automatically generate test cases for statecharts, considering hierarchy and orthogonality, and to synthesise prop-

erties from those test cases generated for the statecharts. These two automatic procedures aims to make easier and cost-effective the tasks of generating test cases and providing formal properties for systems requirements.

Chapter 7

Subjective chapter

In this chapter the author is free to express his own impressions and opinions about the knowledge and experience acquired during this project.

7.1 Learning

During this project, I had the opportunity to learn in greater depth some topics in computer science that were briefly discussed or even not presented in regular courses, such as testing, formal specification and formal verification.

In relation to formal specifications, it was interesting to realise that statecharts can be visually appealing and simultaneously provide accuracy to model scenarios. Although they need to be more spread in the community and industry, statecharts can be applied to real projects.

The concepts learned about testing complemented the practical experience from the internship. In addition, I believe that, due to the contact with testing theory, the execution of my work became more consistent and the maturity level of my professional activities increased. On the other hand, my practical experience in the market also helped me in writing this monograph, since I could see many concepts being applied out of the academic environment.

Finally, it was fun to connect distinct areas (testing, formal specifications, formal verification, sequential mining, logic) to automate processes in the software development cycle.

7.2 Challenges

One challenge was to find a free open-source tool to create statecharts. Initially, we were using a tool that already had test criteria for statecharts, but there were many bugs so we had to abandoned it. Yakindu was a great finding and contributed a lot to the progress of the project.

Since we could not use the initial tool any more, the test case generation for statecharts had to be implemented. The orthogonality feature needed more attention, specially to understanding the semantics behind the diagram modelling concurrency.

7.3 Courses

I believe that the Computer Science curriculum offered by the Institute of Mathematics and Statistics at University of São Paulo gives to students a broad knowledge of many fields

in computer science and a strong mathematical background. From my perspective, though, some topics should be explored in greater details in the program. For instance, it is not offered a course regarding software quality, which is an important set of concepts that are required from students as soon as they start their professional career. The only course that stressed the importance of testing, but not as a main topic, was *Extreme Programming Lab*, which is not even a compulsory course. Moreover, I believe that more techniques and tools related to formal methods, such as formal specifications, could be presented. Among the many courses taken, I consider the ones below more directly relevant for the conclusion of this project:

- **MAC0332 - Software Engineering**

  Comprehension of the activities that take place in the process of software development and their purposes.

- **MAC0414 - Formal Languages and Automata Theory**

  Studied formalisms and algorithms regarding state machines.

- **MAC0239 - Formal Methods in Programming**

  First contact with formal methods and linear time logic.

- **MAC0342 - Extreme Programming Lab**

  The only course in which students were explicitly required to test their implementations. Not only unit tests were written, but also acceptance tests with real clients were performed.

- **MAC0242 - Programming Lab 2**

  More knowledge about Java and object oriented programming were acquired.

Appendix A

Linear Temporal Logic (LTL)

In the next sections we present the syntax and semantics of linear temporal logic based on [16] and [30].

A.1 Syntax

LTL has the following syntax given in the Backus Naur form:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \Diamond\phi \mid \Box\phi \mid X\phi \mid \phi U \phi \mid \phi W \phi \mid \phi R \phi$$

where $p \in A$, the set of propositional atoms.

A.2 Semantics

- **Definition:** A transition system $M = (S, \rightarrow, L)$ is a set of states S endowed with a transition relation $\rightarrow$ (binary relation on S), such that every $s \in S$ has some $s' \in S$, and a labeling function $L : S \rightarrow 2^A$.
- **Definition:** A path π in model $M = (S, \rightarrow, L)$ is an infinite sequence $s_1, s_2, ...$ in S such that, for each $i \geq 1, s_i \rightarrow s_{i+1}$. We write π^i for the suffix starting at s_i.

Let $M = (S, \rightarrow, L)$ be a model and $\pi = s_1 \rightarrow ...$ be a path in M Whether π satisfies an LTL formula is defined by the satisfaction relation $\models$ as follows:

1 $\pi \models \top$

2 $\pi \not\models \bot$

3 $\pi \models p \Leftrightarrow p \in L(s_1)$

4 $\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi$

5 $\pi \models \phi_1 \wedge \phi_2 \Leftrightarrow \pi \models \phi_1$ and $\pi \models \phi_2$

6 $\pi \models \phi_1 \vee \phi_2 \Leftrightarrow \pi \models \phi_1$ or $\pi \models \phi_2$

7 $\pi \models \phi_1 \rightarrow \phi_2 \Leftrightarrow \pi \models \phi_2$ whenever $\pi \models \phi_1$

8 $\pi \models \Box\phi \Leftrightarrow$, for all $i \geq 1, \pi^i \models \phi$

9 $\pi \models \Diamond\phi \Leftrightarrow$ there is some $i \geq 1$ such that $\pi^i \models \phi$

10 $\pi \models X\phi \Leftrightarrow \pi^2 \models \phi$

11 $\pi \models \phi U \psi \Leftrightarrow$ there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, ..., i-1$ we have $\pi^j \models \phi$

12 $\pi \models \phi W \psi \Leftrightarrow$ either there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, ..., i-1$ wwe have $\pi^j \models \phi$; or for all $n \geq 1$ we have $\pi^n \models \phi$

13 $\pi \models \phi R \psi \Leftrightarrow$ either there is some $i \geq 1$ such that $\pi^i \models \phi$ and for all $j = 1, ..., i-1$ wwe have $\pi^j \models \psi$; or for all $n \geq 1$ we have $\pi^n \models \psi$

Bibliography

[1] Yakindu statechart tools. http://www.statecharts.org/. Last accessed in 11/09/2015. 45

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. 1, 8, 9, 10

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. 17

[4] Boris Beizer. *Software Testing Techniques*. 2 edition, 1990. 8

[5] Kirill Bogdanov. *Automated testing of Harel's statecharts*. PhD thesis, Department of Computer Science, University of Sheffield, January 2000. 23, 27, 32

[6] Jonathan P. Bowen, Kirill Bogdanov, John A. Clark, Robert M. Hierons, and Paul Krause. Fortest: Formal methods and testing. 1

[7] Edsger W. Dijkstra. The humble programmer. ACM Turing Lecture 1972. 1, 17

[8] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *2nd Workshop on Formal Methods in Software Practice*, May 1998. 18, 38

[9] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999. 18

[10] L.-H. Eriksson and K. Johansson. Using formal methods for quality assurance of interlocking systems. 16

[11] Formal Methods Europe. Formal methods. http://www.fmeurope.org/?page_id=2. Last accessed in 11/11/2015. 1

[12] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C. Wu., and V. S. Tseng. SPMF: a Java Open-Source Pattern Mining Library. *Journal of Machine Learning Research (JMLR)*, 15:3389–3393, 2014. 38

[13] Irbis Gallegos, Omar Ochoa, Ann Gates, Steve Roach, Salamah Salamah, and Corina Vela. A property specification tool for generating formal specifications: Prospec 2.0. 2, 17, 18

[14] Patrice Godefroid. Combining model checking and testing. 17

[15] David Harel, Amir Pnueli, Jeanette Schmidt, and Rivi Sherman. On the formal semantics of statecharts. In *Proceedings of Symposium on Logic in Computer Science*, pages 55–64, 1987. 1, 3, 4, 6

[16] Michael Hauth and Mark Ryan. *Logic in computer science. Modelling and reasoning about systems*. Cambridge University Press, 2 edition, 2004. 1, 57

[17] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Inc, 2 edition, 1998. 3, 4

[18] Lu Luo. Software testing techniques - technology maturation and research strategy. Technical report, Institute for Software Research International, Carnegie Mellon University, Pittsburgh,USA. 1, 8

[19] Nizar R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43, 2010. 19, 20

[20] José Carlos Maldonado, Ellen Franciane Barbosa, Auri M. R. Vincenzi, Márcio Eduardo Delamaro, Simone do Roccio Senger de Souza, and Mario Jino. Introdução ao teste de software, 2004. 10

[21] José Carlos Maldonado, Márcio Eduardo Delamaro, and Mario Jino. *Introdução ao Teste de Software*. Elsevier, 2007. 14

[22] Tom Mens. Yakindu statechart tools video tutorials. http://statecharts.org/videos.html. Last accessed in 11/18/2015. 5

[23] Stephan Merz. Model checking: A tutorial overview. 18

[24] NASA Langley Formal Methods. What is formal methods? http://shemesh.larc.nasa.gov/fm/fm-what.html. Last accessed in 11/11/2015. 1

[25] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing*. John Wiley & Sons, Inc, 3ž edition, 2012. 8

[26] NASA. Testing vs. model checking. http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/testing_vs_model_checking#no1. Last accessed in 11/07/2015. 17

[27] Jian Pei, Behzad Mortazavi-Asl, and Umeshwar Dayal. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16, 2004. 19, 20, 21, 38

[28] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. 19, 20

[29] Jeff Tian. *Sofotware Quality Engineering*. John Wiley & Sons, 2005. 17

[30] Wikipedia. Linear temporal logic. https://en.wikipedia.org/wiki/Linear_temporal_logic. Last accessed in 11/11/2015. 1, 57

[31] Érica Ferreira de Souza. Geração de casos de teste para sistemas da área espacial usando critérios de teste para máquinas de estados finitos. Master's thesis, Instituto Nacional de Pesquisas Espaciais - INPE, São José dos Campos,Brazil, February 2010. 14, 15, 16