Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Rafael Mota Gregorut

Título da monografia se for longo ocupa esta linha também

São Paulo
Dezembro de 2015

Título da monografia
se for longo ocupa esta linha também

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Orientadora: Prof. Dr. Ana Cristina Vieira de Melo

São Paulo
Dezembro de 2015

Abstract

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave).

Keywords: keyword1, keyword2, keyword3.

Contents

Chapter 1

Introduction

bla bla bla

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.

This project proposes a study on the set of test cases generated from specifications in order to guide properties definitions. First, a technique to automatically generate test cases from the specification is presented. Then, all test cases are observed and a technique to synthesize properties from the acquired tests is proposed.

1.2 Goals

- Implement an algorithm to automatically generate test cases based on statechart specifications
- Implement a technique to automatically 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, synthesize properties to be used by model check programs
- Develop a tool to help these tasks

1.3 Organization

In chapter 2, we present the concepts studied to make this project possible. In chapter 3, we describe an 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 case study demonstrating the usage of the implemented tools. Our conclusions regarding this project can be found in chapter 6. Information about the tool used to create the statechart models are in apendix A. Finally, the syntax of the linear temporal logic, the logic used in the generated formal properties, can be found in apedix B.

Chapter 2

Concepts

2.1 Statecharts and Automata

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

Statechats are a type of formal software specification based on finite states machines (FSM) specially used in complex systems modeling, such as reactive systems and were defined in [11]. 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 and history.

2.1.1 Nondeterministic finite automata

Definition[12]: 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 tripe $(q, a, p) \in \Delta$ 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) or (q, e, p). In the later case, no input is read. A configuration of M is an element of K^* and the relation $\vdash$ (yields one step) between two configuarions is defined as: $(q, w) \vdash (q', w') \Leftrightarrow$ there is a $u \in \Sigma \cup e$ such that $w = uw'$ and $(q, u, q') \in \Delta$.

Further information and formalism regarding finite automata can be obtained in [12].

A nondeterministic finite automaton example

Find below an example of a nondeterministic finite automaton extracted from [12].

- $K = \{q_0, q_1, q_2\}$

- $\Sigma = \{a, b\}$
- $\Delta = \{(q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_0), (q_2, e, q_0)\}$
- $s = q_0$
- $F = \{q_0\}$ is the set of final states

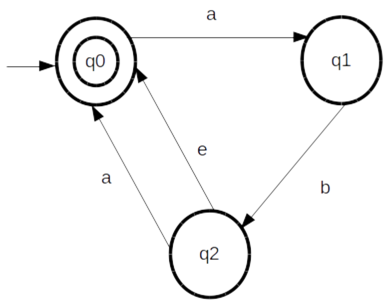

Figure 2.1: *A nondeterministic finite automaton that accpets language* $L = (ab \cup aba)^*$.

2.1.2 Statechart models

A statechart model can be considered a extended finite state machine. The syntax of statechart is defined over the set of states, transitions, events, actions, conditions, expressions and labels [11]. 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 occurence of that transitions
- Expressions are composed are variables and algebraic operations over them
- A label is a pair made of an event and an action that is used to label a transition

Furthermore, it is worth noting that a statechart model possess other features, described over the next sections of this chapter, that do not appear in a common finite state machine. These extra resources are useful, for instance, to model concurrency and different abstraction levels in a more practical way.

Orthogonality

More than one state may be active at the same time in a statechart, which 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, parallel regions inside the state *Clock*: $r1$ and $r2$. At the same moment, then, states *Display* and *AlarmOff* can be active.

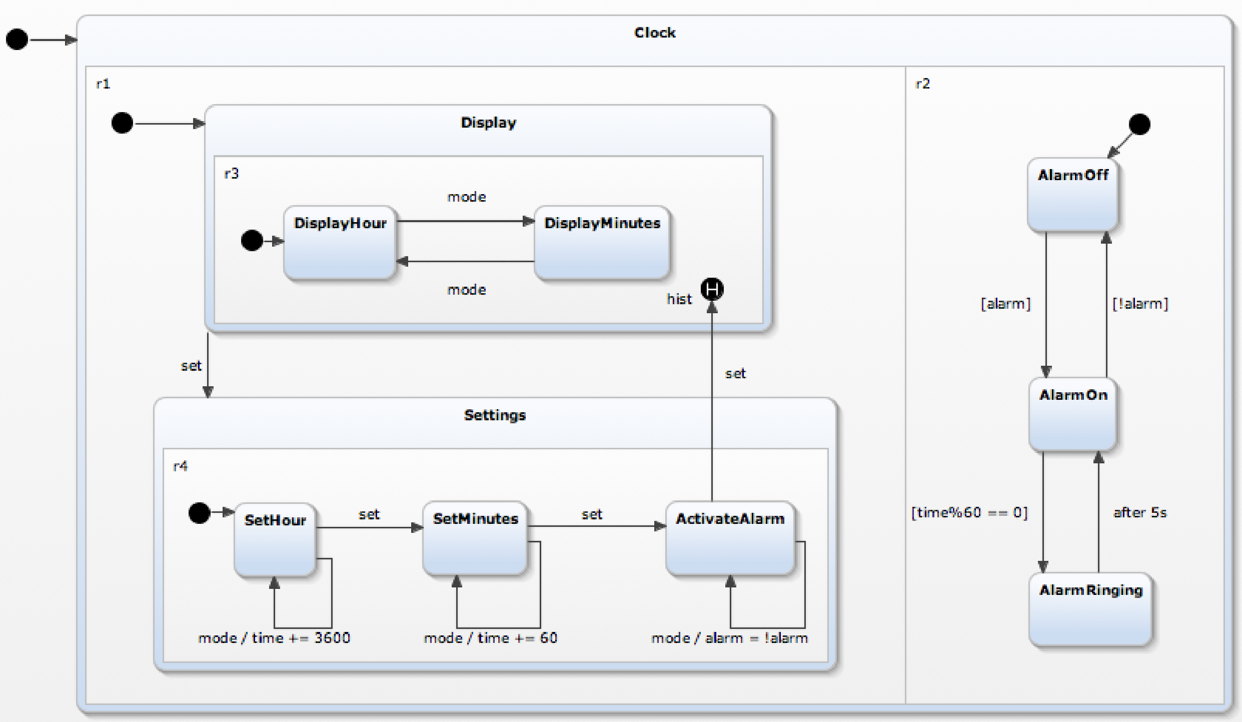

Figure 2.2: *Statechart example. A clock model.*

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 sub-states of $Clock$. $Display$ also contains the sub-states: $DisplayHour, DisplayMinutes$ and $hist$. And the states $SetHour, SetMinutes$ and $ActivateAlarm$ are inside state $Settings$.

Guard conditions

In a transition, a guard condition is always verified before the change of states take place. If the condition is satisfied, in other words, e expression returns true, the transition will happen. Otherwise, that transition is not allowed to happen. A expression in a guard condition involve variables and operations. It is possible to check whether a state was entered for example. In figure 2.2, the transition from $AlarmOff$ to $AlarmOn$ is guarded by the condition $[alarm]$, meaning that the trasition will only happen if the value of the variable $alarm$ is $true$.

Broadcasting

Besides an input event, a transition might have a action that is triggered when the transition is done. An action may cause another transition to happen, that is called broadcasting. This features makes it possible that chain reactions occurr in a statechart model. Consider 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 satified. Therefore, not only the transition starting at *ActivateAlarm* happened when *mode* occured, but also the transition between *AlarmOff* and *AlarmOn*. The change of the value of variable *alsarm* 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. Considering 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 occurance of *set* we will be directed to the history state *hist*. Since the last active sub-state in *Display* was *DisplayMinutes*, we will actually be directed to *DisplayMinutes*.

A statechart and its correponding automaton

It is possible to get an automaton from a statechart by flattening the statechart. The hierarchy and cocurrency 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 [11]. In figure 2.3 we a statechart and in figure 2.4 we have the flat version that would correspond to an automaton.

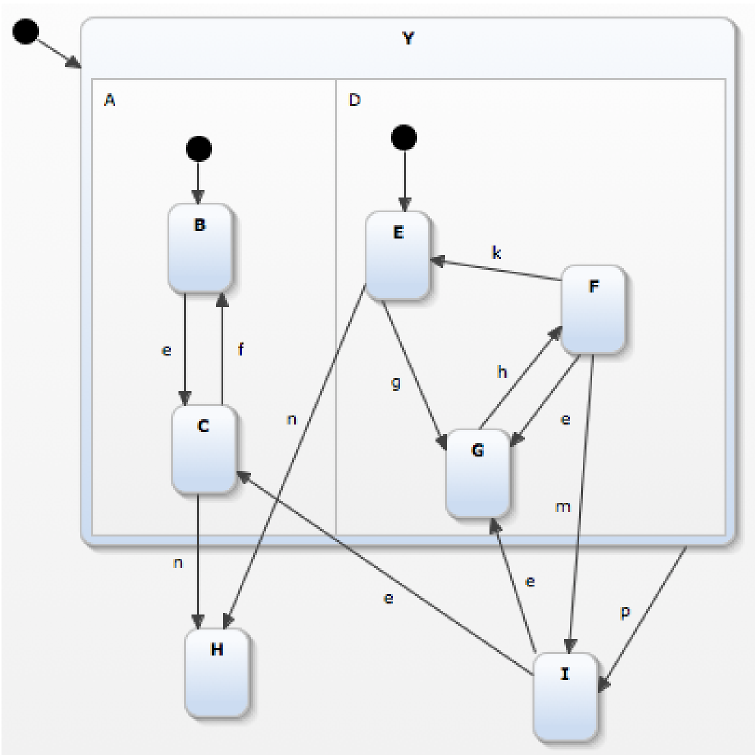

Figure 2.3: *A statechart model with hierarchy and concurrent regions*

To flat a statechart and get the corresponding automaton, we need to pass transitions from the composed states to their sub-states to eliminiate the hierarchy. To remove orthogonality, we need to do the product of the states in each concurrent region. Such operation causes will cause state and transition explosions in the automaton if the original statechart model has many concurrent states. Besides, in the statechart we can make usage of guard transitions and broadcasting, enriching the model with more information. Note that in the

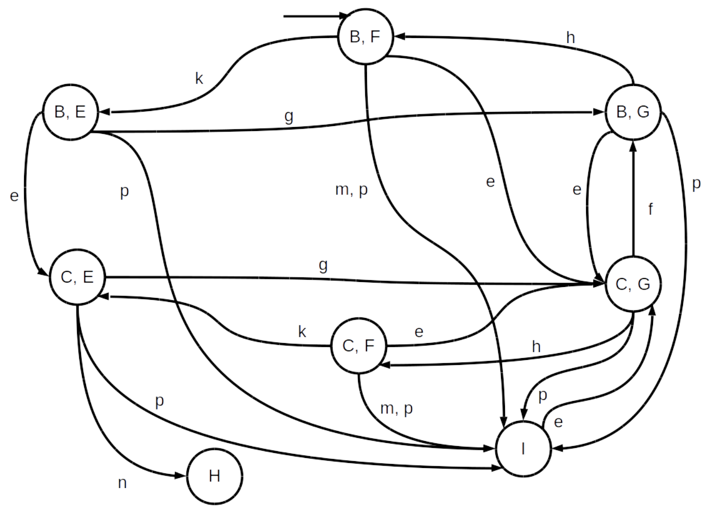

Figure 2.4: *The statechart from 2.3 after flattening to the corresponding automaton*

automaton, the initial state is the product of the initial states from each parallel region in the statechart.

2.2 Tests

Since code has been written, programs have been tested. 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 [13]. Therefore, it constitute an important area in software engineering.

Testing evolved from an activity related debugging code to way of checking specification, design as well as implementation[13]. It can be considered a process not only to detect bugs, but to also prevent them [3].

2.2.1 Testing goals

In a organization, the goals of testing vary depending on the level of maturity of the team [1]: At a more inexperienced approach, testing could be viewed as the same as debugging the code. A further step would be to consider testing as the process to make sure that a software does what it is supposed to do. But, in this case, if a team runs a test suite and every test case passed, should the team consider that the software is correct or could it be that the test cases were not enough? How does the team decide when to stop?

A different perspective, then, would be to understand testing as the process of finding erros. Therefore a successful test case would be the one that indicated an error in the system [18]. Altough descovering unexpected results and behaviour is a valid goal, it might put tester and developers in an adversarial relationship, which certainly demages 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. [1]. 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 valdilated, 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 procress, even though their execution will only be possible after some part of the code is implemented.

V-Model

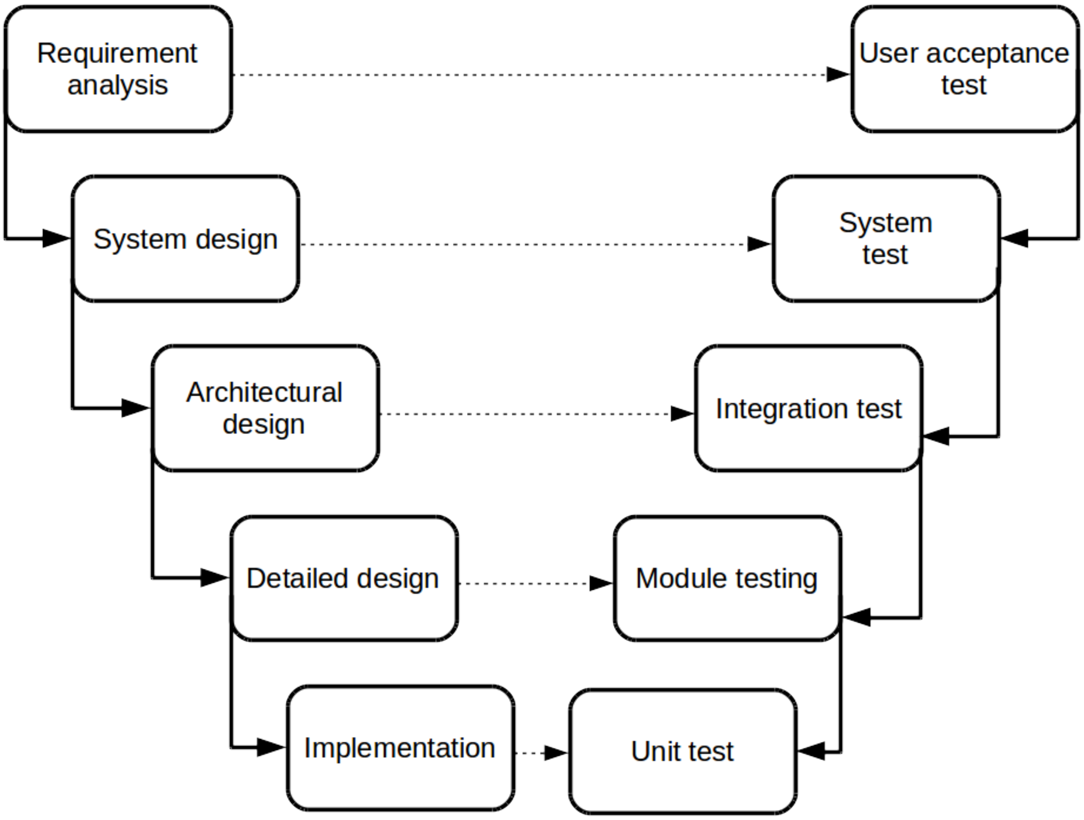

Figure 2.5: *The V-model*

The V-model in figure 2.5 associates each level of testing to a different phase in the development process. This model is typically viewd as an extansion of the waterfall methodology, but it does not mandatorily implies the waterfall approach since the synthesis and anlysis activities generically apply to any development process [1].

- requirement analysis phase: Customer's needs are registered. **User acceptance tests (UAT)** are constructed to validated that the delivered system meets the customer's requirements.
- System design phase: Technical team analyzed the captured requirements and study the possibilities to implement them and document a technical specification. **System**

tests are designed at this stage assuming that all pieces work individually and checks if the system works as a whole.

- Architectural design phase: Specify interface relationships, dependencies, structure and behaviour of subsystems. **Integration tests** are developded, 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 one of them with the corresponding behaviour specified so that the programmer can code. **Module testing** is designed to check eahc module individually.
- Implementation phase: Code is actually produced. **Unit tests** are designed to test every smallest unit of code, such a method, can work correclty when isolated.

2.2.3 Functional and Structural tests

Testing techniques can be divided in functional and structural categories.

Functional technique (black box)

Functionalities described in the specification are considered to create the test cases. It is necessary to study the behaviour and functionalities presented, develop the corresponding test cases, submit the system to the test cases and analyze the results observed comparing them to what was exected according to the description in the specification. Examples of criteria in this technique[15]:

- **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 a element in each class are a representative of the whole class. Using this criterion, the test cases can be executed sistematicaly according to the requisites.

- **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 descriptive way, the test cases developed using the functional technique can be informal and not specific enough, which makes it difficult to automate their execution and human intervention becomes necessary. However, this technique only requirers that requisites, input and corresponding output are identified. Thus, it can be apllied during several testing phases, such as integration and system.

Structural technique (white box)

The structure of the code implementation is considered to create the test cases. In this technique, the program is represented by an oriented graph of flow control, in which each vertice corresponds to a block of code, a statement in the code for instance, and each edge corresponds to a trasition between the blocks. The criteria related to this technique are generally concerned with the coverage of the program graph, such as [1]:

- **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 subpath 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. In sutiation might occurr if the use scenarios provided by the customer are converted to paths in the graph and the team wishes to test them.

Tests obtained via 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 customer's requirements, since the specified requisites 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 analyzed. Instead, the specifications with the requisites 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 composed by 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 functionality defined, use the equivalence class partitioning technique described in 2.2.3 to select the input representatives; enumarate the steps necessary to activate the functionality; and add the expected outputs. Note that requisites to each step can 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 functionalities 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. In other words, 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 functionality, the team can easily keep track of how many defects each functionality has, what the most problematic scenarios are as well as what the most critical bugs are.

But manually writting test cases is not an simple task. The engineer must read through all the specification, identify functionalities and the possibly many scenarios they can be executed. It requires experience and time to project test cases that have more probability to find defects and, therefore, will decrease the risk of using the application. We present in chapter 3 a method to automaticaly generate test cases for well defined statechart specifications.

To ilustrate, one test case example to test signing in functionality in a website could be the following:

Step	Requisites	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 functionality:

Step	Requisites	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."

Use cases vs Test cases

More detalied use cases provide a series of steps to perform flows related to a determined functionality. Each step might contain the description of system behaviour and user actions. There are also preconditions so that the use case be considered for execution and an actor is associate with the case. Besides, one use case might describe more than one flow for the functionality by extending the basic flow. Below, we can find an use case example of signing in of 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 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 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 error messages

In a test case, however, one step have to specify one single action and its corresponding consequence in the system. We can add precondtions to each step in a test case, even if the precondition is 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.

An use case serves as an guide for during the implementation process used by developers. It is a way to understand how the functionality 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 funcionalities relevant to the software. Considering that such model correct, is readable by a machine and its interpretation is well defined, one can use it to automatically generate functional test cases [16]. This technique, in which test cases are automatically derived from a model, is called Model Based Test.

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. Some of the them are described below and with further details in [23]:

- **All transitions**

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

- **All simple paths**

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

  It is important to note that even though this two previous criteria 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 discribed 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, the finite state machine must be deterministic, complete, strongly connected and minimal.

$\beta - sequence$	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".[23]*

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 including t is generated. That sequence is called $\beta - sequence$.

We can then concatenate SD to 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.

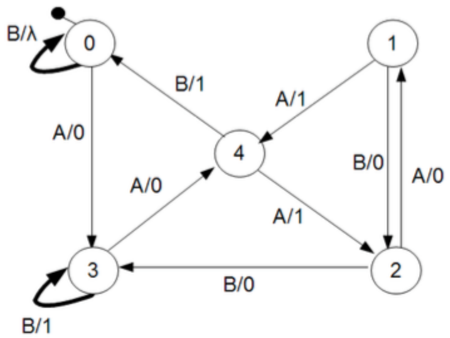

Figure 2.6: *Automaton extracted from [23]*

- **Unique Input-Output**

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

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

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.[23]*

$\beta - sequence$	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.[23]*

First, for each state, an 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 the previous criterion.

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

For the automaton in 2.6, table 2.2 shows the possible UES for each state. In table 2.3 we the whole set of $\beta - sequences$ and their concatenation with the respective UES. Both tables can be used to ilustrate the Unique Input-Output criterion.

2.4 Model checking

Formal methods refers to the use of precise logical and mathematical methods to reason about properties of the system [7]. They are used in software and hardware engineering fields with formal specification and formal verification techniques aiming to contribute 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[22].

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[2]. Typically, there is a hardware or software system specification containing the requirements, and we wish to verify that certains properties, such 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 generally 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, "program testing can at best show the presence of errors but never their absence". On the other hand, if there is a violation of a given specification, it will be found by model checking, which is supposed to be a rigorous method that exhaustively explores all possible behaviours of the system under consideration [19]. This is the main feature that distinguishes testing and model checking: the last can prove the absence of bugs[10].

According to [2], the model checking process can be divided in the following phases:

1 **Modeling 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 analyze 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 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 fixes. 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 modeling 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[2]. Hence, a property specifies a certain behaviour of the system that is being considered. One has to manually read through the specification and manually definy 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 to write the formal property specification must have a mathematical background and knowledge of the specification language [9]. 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. Translating system requirements to formal properties is not easy.

The classic example for property definition is the absence of deadlocks, but properties can also specify safety protocols[17], 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 the following LTL property to verify the requirement:

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

The LTL syntax can be found in apendix B.

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 [9]. To assist this process, a set property specification patterns were proposed in [5]. 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 projects [6].

A hierarchy was establised to facilitate browsing through the patterns and picking the most appropriate one to one's need. The first level of the hierarchy is composed by 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 further classified in:

- 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 [14]. 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 [21]. In other words, if a user inputs a percentage p and a sequence database D, then the mining will return the set of patterns that are present in at least $p\%$ of the data-sequences. The formal definition can be found in [14] and in [20].

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 [14].

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 $l_{i_1}, ..., l_{i_m}$ 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 a ecommerce application, for instance, we can have the set of items $I = a, b, c, d, e, f$ representing products that can be purchased. The ocurrence 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 frequen pattern, indicating that 90% of the users who visit 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 taks, but they generally differ in two aspects[14]:

- 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, 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 all infrequent. An example for the category would be the GSP[21].

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 count for each candidate and test it at iteration of the algorithm.

Generally, algorithms in this category generate an explosive number of candidate sequences, consuming a lot of memory. Besides, methods such as *GSP* generates a combinatorially explosive number of candidates when mining long sequential patterns. That may be the case of a DNA analysis application, in which many patterns are long.

In addition, since they need to check at each iteration for the support count, multiple scans of the database ate performed, which requires a lot of processing time and IO cost. In general, to find a sequential pattern of length l, the a priori-based method must scan the database at least l times. When long patterns exist, this characteristic will cause a non-trivial cost.

Pattern-growth algorithms

Pattern-growth algorithms try to use a certain data structure to prune candidates early in the mining process. Besides, the search space is partitioned for efficient memory management. *PrefixSpan*[20] is an example of such algorithm and, since it was used in our implementations, we will provide more information about it in this subsection.

The general idea behind *PrefixSpan* is as follows: Instead of repeatedly scanning the entire database and 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[20].

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 than apriori-based algorithms, in PrefixSpan does not generate or test candidate sequences. Patterns are grown from the shorter ones. Besides, projected databases keep shrinking, which is relevant in practice because usually, only a small set of sequential patterns grow 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, [20] proposes a technique called *pseudo partition* that may reduce the number and size of projected databases.

Chapter 3

Implementation of test case generation for statecharts

In this project, we implemented the test case generation for statecharts based on the criteria described in [4]. We test every transition by visiting every state and trigger events for all transitions that start in it.

3.1 Test cases for simple statecharts

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

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 [4].

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

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

  Set setC = new Set();

  Path emptyPath = "";

  List visited = new List();

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

```
//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;
  }
```

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 exercides at least once during testing.

Consider the transition $t = (s, l, q)$, where s is the original state, l is the event label that triggers t and q is the destiny state. Previously, we computed that s has coverage path p such that $p \in C$ and p is a sequence of label. The test case TC to t would be the concatenation of the event label l to the end of p expecting to get to state q. The process is repeated to every transition in the statechart. Find below the algorithm for the test case 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 following example in figure 3.1 where we model the verifications in a purchase flow of a telco ecommerce. The flow is done by a user who wishes to change their cellphone plan. They will be able to change plan if their not employees from the telco company and are not committed to a loyalty contract.

Hierarchy and orthogonality are not found in statechart 3.1. Hence, we can apply the the technique just presented.

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

We start at the inital state *Unlogged User*. Since it is the initial state, its coverage path

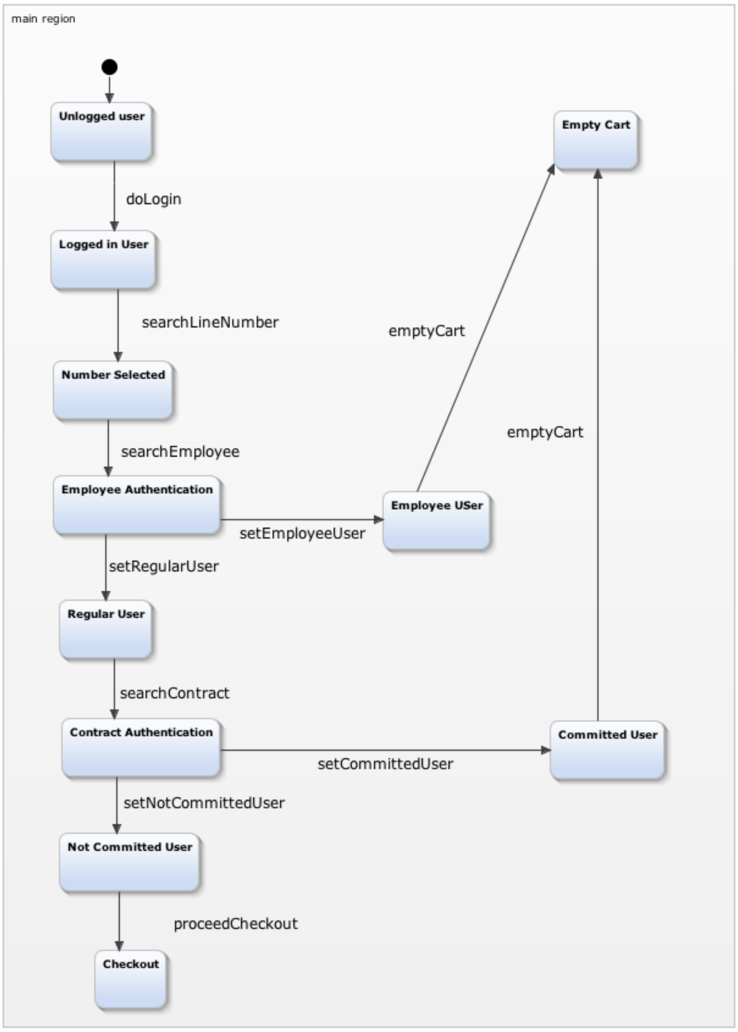

Figure 3.1: *A plan change flow statechart in a telco ecommerce*

is the emtpy string, denoted by e.

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 concatenate the coverage path of the previous state to the transition taken.

When construction of C is done, we have the following covage 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

Notice the empty string e is present due to the unlabeled 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 exercided at least once during testing and that they go to their appropriate destiny states, we need the following test cases:

- Test case for ***setEmployeeUser***

  Path: *e doLogin searchLineNumber searchEmployee setEmployeeUser*

  Expected state: *Employee User*

- Test case for ***setRegularUser***

  Path: *e doLogin searchLineNumber searchEmployee setRegularUser*

  Expected state: *Regular User*

Note that the path to test is the coverage page of the origin state concatenated with the label of the tested transition. The analogous is done for all other transitions in the model.

3.2 Test cases for complex statecharts: hierarchy

In this section, the test cases generated will be obtained from statecharts that have hierarchy: 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 a the superstate of b and b the substate of a.

One way to deal with hierarchy is to eliminate it from the model by flattening the statechart as shown in 2.1.2. The statechart would become an automaton and the thecniques for simple statecharts explained in the previous section could be used to generate test cases.

But, the approach taken in this project, as in [4], was to keep the structure of the statechart and create the test cases incrementally.

Similarly to the previous simpler case, for statecharts with hierarchy we still 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 consideration substates to cover them as well. It is important to note that we considered only statecharts that do not have transitions between different hierarchy levels.

When we get to a state, we should 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 the begining of each coverage path of the substates. Then, the coverage path of the super state should be removed from C and the paths to the substates will be kept in C instead. Besides, consider the case in which 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 needs some changes. Regarding the pseudocode presented in the previous section, we need to modify the recursive function:

```
//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 + Δs + t.getLabel());

      } else {
```

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

        }

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

    return setC;
  }
```

After the set C is complete, we need to in fact create the test cases based on every transition that leaves each state. In states that do not have substates and whose coverage did not pass by a superstate, the process is the one presented in the previuos section.

In case, the state's coverage path p went through a superstate, we need to expand the path with the coverage paths of the subtates. 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. Follows the algorithm for the expansion:

```
//Pseudocode for an expansion function
//Receives the original, not expanded, path and the super state is 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(Δsuper, subPath);

    expansionResults.add(expanded);
  }

  return expansionResults;
}
```

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 to t, we will 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);
    }
```

```
    }
  }
```

The pseudocode to create the final test cases is the same as the one presented in 3.1.

Let's take as an example the statechart in figure 3.2. It models an application that receives order files in a specific format and converts them into an well formatted xml. Each order file contains several lines, and each line contains a product and its quantity. The application also has an integration layer, that will receive the xml file and then send it to the management system.

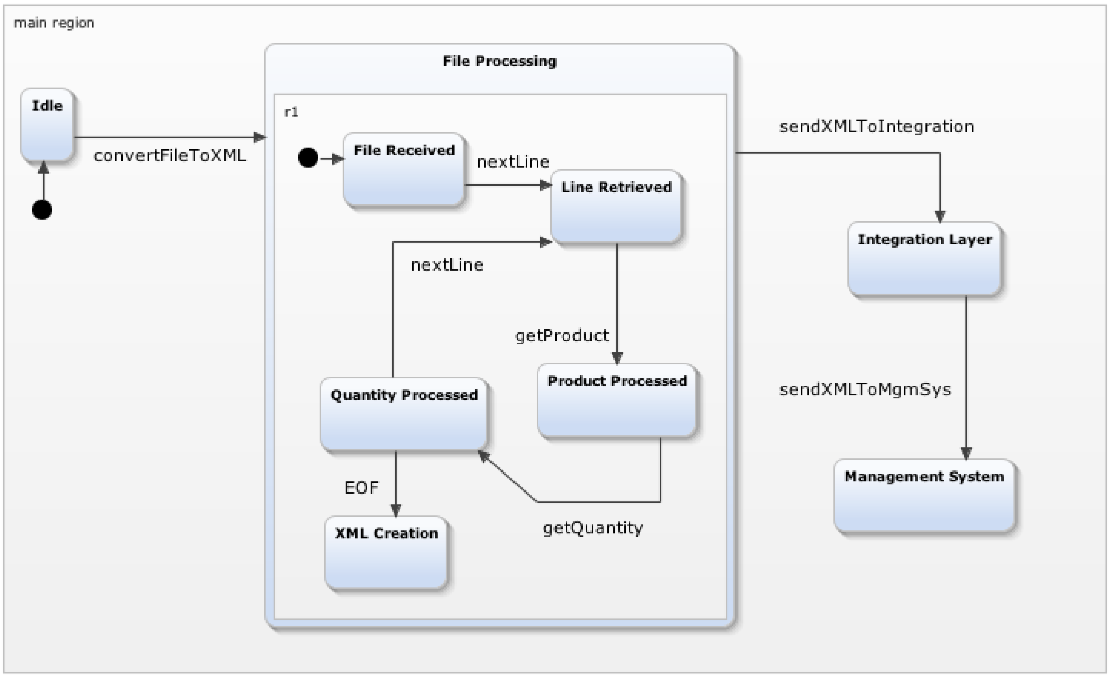

Figure 3.2: *Statechart for order file processing and transference*

In figure 3.2, to contruct 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 unlabeled transition leaving the initial state.

When creating the coverage path for state *File processing*, we notice that it actually is superstate. So, instead, 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*. We should add the following paths 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

Notice that the coverage path for the superstate *File processing* is removed from C, because we are considering its substates. Therefore, the testcases that would be created based on the superstate will be created based on the substates. Besides, the empty string e is added twice in each of these paths. That is beacuse they pass through two initial states with unlabeled transitions: the first one is the general initial states, and the second one is the initial state inside *File processing*.

We still need to cover states *Integration layer* and *Management system*. Observe that to cover these states, we need to pass by *File processing*, a superstate. Therefore, in their coverage path, we need to use a specific notation to guide the later expansion with the substates coverage paths. We will use the notation $\Delta_{File\ processing}$ to indicate that, when creating the test cases, we need to expand the path considering the coverage of *File processing*'s substates. Thus, the coverage paths for *Integration layer* and *Management system* are as follows:

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

At this point, that we have the complete set C, we start to in fact create the test cases. As an example, we will examine transitions *getProduct, sendXMLToIntegration* and *sendXMLToMgmSys* more closely.

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

- Test case for ***getProduct***

  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* making usage of p. We should append $sendXMLToIntegration$ to each p in order to obtain the test paths:

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

  Path: *e convertFileToXML e sendXMLToIntegration*

  Expected state: *Integration layer*

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

  Path: *e convertFileToXML e nextLine sendXMLToIntegration*

  Expected state: *Integration layer*

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

  Path: *e convertFileToXML e nextLine getProduct sendXMLToIntegration*

  Expected state: *Integration layer*

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

  Path: *e convertFileToXML e nextLine getProduct getQuantity sendXMLToIntegration*

  Expected state: *Integration layer*

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

  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 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***

  Path: *e convertFileToXML e sendXMLToIntegration sendXMLToMgmSys*

  Expected state: *Integration layer*

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

  Path: *e convertFileToXML e nextLine sendXMLToIntegration sendXMLToMgmSys sendXMLToMgmSys*

  Expected state: *Integration layer*

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

  Path: *e convertFileToXML e nextLine getProduct sendXMLToIntegration sendXMLToMgmSys*

  Expected state: *Integration layer*

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

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

  Expected state: *Integration layer*

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

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

  Expected state: *Integration layer*

Notice that, in each case, $\Delta_{File\ processing}$ in *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 possess orthogonality, in other words, states in concurrent regions.

One fisrt method to generate test cases dealing with orthogonality is to eliminate it by flattening the statechart as explained in 2.1.2. The elimation 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 [4].

To avoid state and transition explosion and still be able to cover all states and test all transitions, [4] offers the alternative to refine the concurrency requisites. In this project, we chose 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 sequences of transitions to occur. A chain reaction would be an example such a sequence and would be expected by the test cases listed by this implementation.

We first compute the covage 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 next pseudocode, we consider that substates are in a region inside the superstate. To apply the pseudocode for statecharts with hierarchy discussed in the previous section (3.2) we should consider that the superstate have only one internal region, which will contain the substates. In statecharts with concurrency, the concurrent elements would be indiferent regions inside a superstate; figure 3.3 serves as an example. Therefore, the solution can be applied to states with orthogonality as well as to ones with hierarchy. Find below the pseudocode for the contruction of 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;
  }
```

Next, we again need to generate the test cases for each transition of the model. This process is the same as the one described in the case with hierarcy since concurrent states are inside a super state. The algorithm for this phase can be found in pseudocode format in 3.1.

To ilustrate this case, we can look at the example provided in figure 3.3. It models an online store in which the administrator is allowed to execute two jobs: one to clear all reservations in every product (region *Clear reservations job* in the figure) and on 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 whishes, thus their regions are modeled in a concurrent way in the statechart.

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* region are inside state *Stock update email*, hence we need to apply the algorithms presented in 3.2. Note that since the coverage path to *Stock update email* is just the empty string e, it will not have a great impact in 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

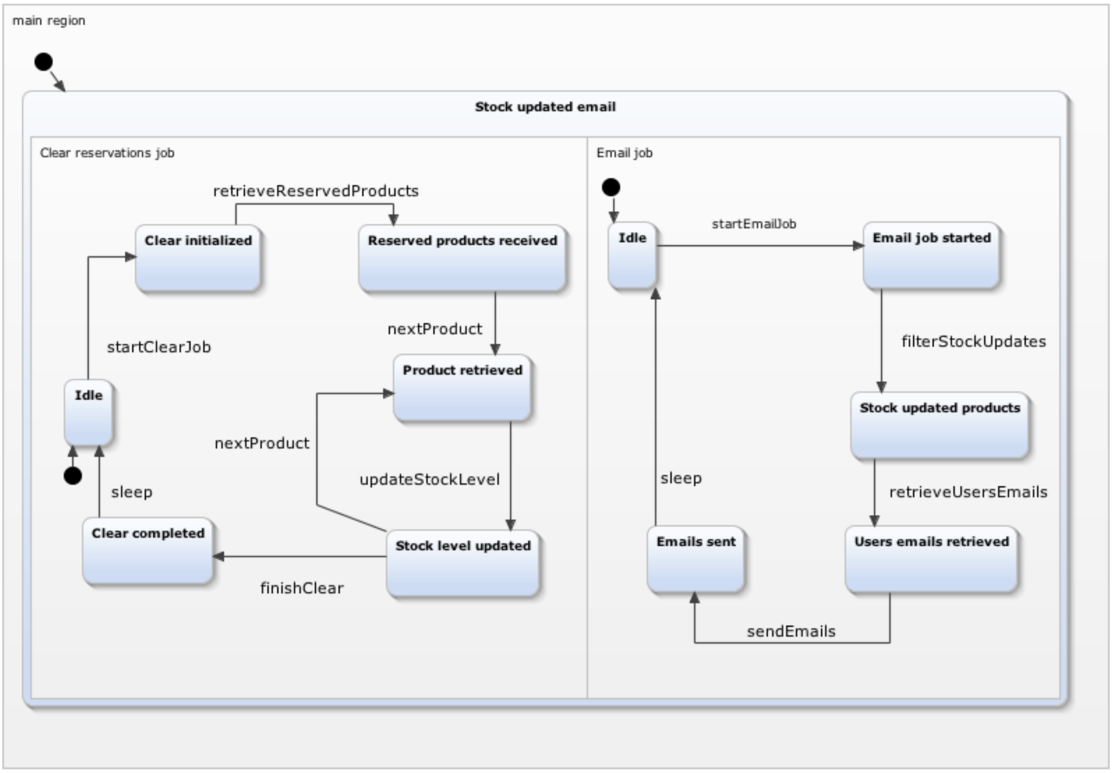

Figure 3.3: *Statechart model for concurrent jobs: clear reservations job and send email job*

Secondly, we can construct the coverage paths for substates in region *Email job* by an analogous process. For instance, the coverage path for *Email sent* would be:

State	Coverage path
Email sent	e e startEmailJob filterStockUpdates retrieveUsersEmails sendEmails

The generation of the test cases, then, is similar to the one presetend in the previous 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 3.3, because we don not have a state after *Stock updated email*, no expansion will be needed.

To demonstrate, we will consider the test cases for transitions *nextProduct*, from state *Stock level updated*, and *sleep*, from state *Email sent*. We need to concatenate these transition labels to the end of 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***

  Path: *e e startClearJob retrieveReservedProducts nextProduct updateStockLevel nextProduct*

  Expected state: *Product retrieved*

- Test case for ***sleep***

  Path: *e e startEmailJob filterStockUpdates retrieveUsersEmails sendEmails sleep*

  Expected state: *Idle*

Chapter 4

Property extraction from test cases

In this chapter we propose a technique to automatically create 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 fragment of them. In order to do so, we will apply the mining concepts and algorithm presented in section 2.5 to extract patterns, which will later be used to derive the properties.

4.1 Test case mining

In chapter 3, we presented an method to automatically generate test cases for 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 us to.

The test path p is basically a sequence of consecutive events. Therefore, it is possible to analyze it with the concepts and notation shown in section 2.5. Let's say that p is composed by 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 sequence database. Hence, we are able to apply sequential pattern mining algorithms, such as *PrefixSpan* (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 role during testing, due to the fact that they are the ones that are stressed the most. 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 also indicates which event subsequences most test cases rely on, so a defect in any of them would block a considerable amount of test case execution. Consider 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 orders, 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 analyzed, it provides subsequences that 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 implementaion, we used the *Sequential Pattern Mining Framework* (*SPMF*)[8] to perform the test case mining. *SPMF* is an open-source data mining library written in Java, specilized 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 instance, as well. For the purpouse of this project, we chose the provided *PrefixSpan* algorithm due to the empirical analysis presented in [20] demonstrating that it would be more efficient than other classic sequential pattern mining algorithms such as *GSP*.

The algorithm receives as input the sequence database and the minimum support value as the user wishes. It then computes the most frequent subsequence patterns, which are internally stored and used during the creation of the formal properties. Note that our implementaion 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 properties generation.

4.2 Generation of properties from most frequent subsequences

4.3 Generation of properties for specific events

Chapter 5

Case study

Chapter 6

Conclusion

Texto texto[1].

[1]Exemplo de referência para página Web: www.vision.ime.usp.br/~jmena/stuff/tese-exemplo

Appendix A

Yakindu Statecharts Tool

Texto texto.

Appendix B

Linear Temporal Logic (LTL)

Texto texto.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. 7, 8, 9

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. 15, 16

[3] Boris Beizer. *Software Testing Techniques*. 2 edition, 1990. 7

[4] Kirill Bogdanov. *Automated testing of Harel's statecharts*. PhD thesis, Department of Computer Science, University of Sheffield, January 2000. 21, 25, 30

[5] 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. 17

[6] 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. 17

[7] L.-H. Eriksson and K. Johansson. Using formal methods for quality assurance of interlocking systems. 15

[8] 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. 34

[9] Irbis Gallegos, Omar Ochoa, Ann Gates, Steve Roach, Salamah Salamah, and Corina Vela. A property specification tool for generating formal specifications: Prospec 2.0. 16, 17

[10] Patrice Godefroid. Combining model checking and testing. 16

[11] 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. 3, 4, 6

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

[13] Lu Luo. Software testing techniques - technology maturation and research strategy. Technical report, Institute for Software Research International, Carnegie Mellon University, Pittsburgh,USA. 7

[14] Nizar R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43, 2010. 18

[15] 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. 9

[16] José Carlos Maldonado, Márcio Eduardo Delamaro, and Mario Jino. *Introdução ao Teste de Software*. Elsevier, 2007. 13

[17] Stephan Merz. Model checking: A tutorial overview. 16

[18] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing*. John Wiley & Sons, Inc, 3ž edition, 2012. 7

[19] 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. 16

[20] 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. 18, 19, 34

[21] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. 18, 19

[22] Jeff Tian. *Sofotware Quality Engineering*. John Wiley & Sons, 2005. 15

[23] 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. 13, 14, 15