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

Resumo

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).

Palavras-chave: palavra-chave1, palavra-chave2, palavra-chave3.

Abstract

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa.

Keywords: keyword1, keyword2, keyword3.

Contents

Chapter 1

Concepts

1.1 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 [4]. 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.

1.1.1 Nondeterministic finite automata

Definition[5]: 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 [5].

A nondeterministic finite automaton example

Find below an example of a nondeterministic finite automaton extracted from [5].

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

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

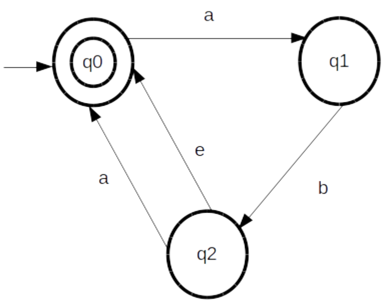

Figure 1.1: *A nondeterministic finite automaton that accpets language* $L = (ab \cup aba)^*$.

1.1.2 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 [4]. 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 1.2, parallel regions inside the state $Clock$: $r1$ and $r2$. At the same moment, then, states $Display$ and $AlarmOff$ can be active.

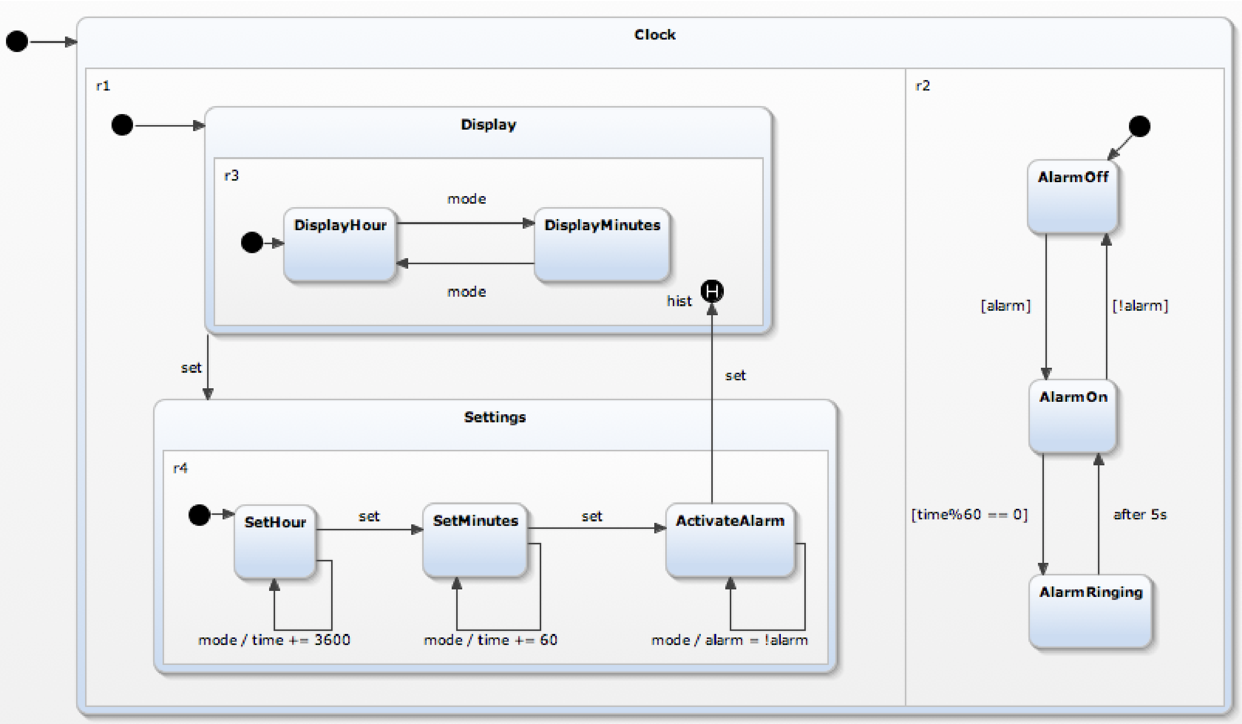

Figure 1.2: *Statechart example. A clock model.(@@@@@ CITAR REFERENCIA DO YAKINDU)*

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 1.2, 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 1.2, 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 1.2, 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 1.2, 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 [4]. In figure 1.3 we a statechart and in figure 1.4 we have the flat version that would correspond to an automaton.

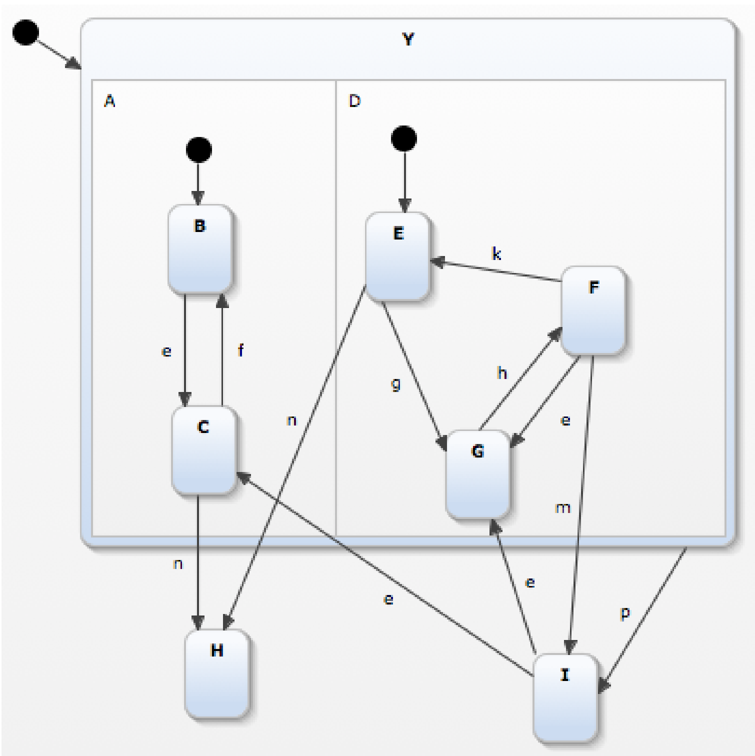

Figure 1.3: *A statechart model with hierarchy and concurrent regions*

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

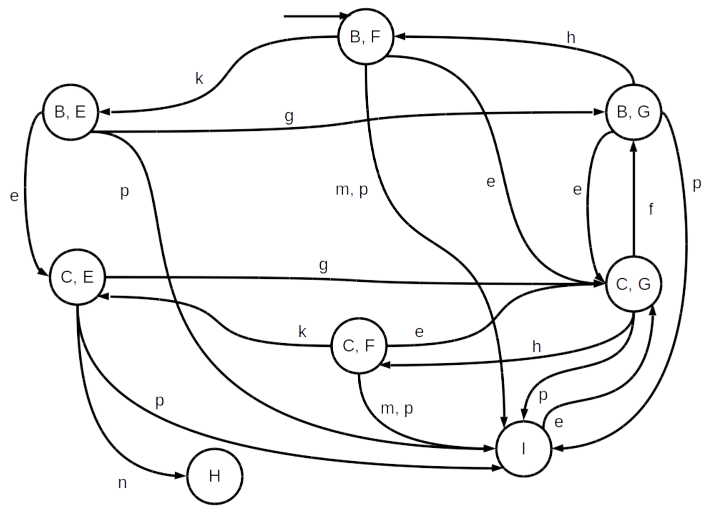

Figure 1.4: *The statechart from 1.3 after flattening to the corresponding automaton*

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

1.2 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 [6]. 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[6]. It can be considered a process not only to detect bugs, but to also prevent them [2].

1.2.1 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 [9]. 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.

1.2.2 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

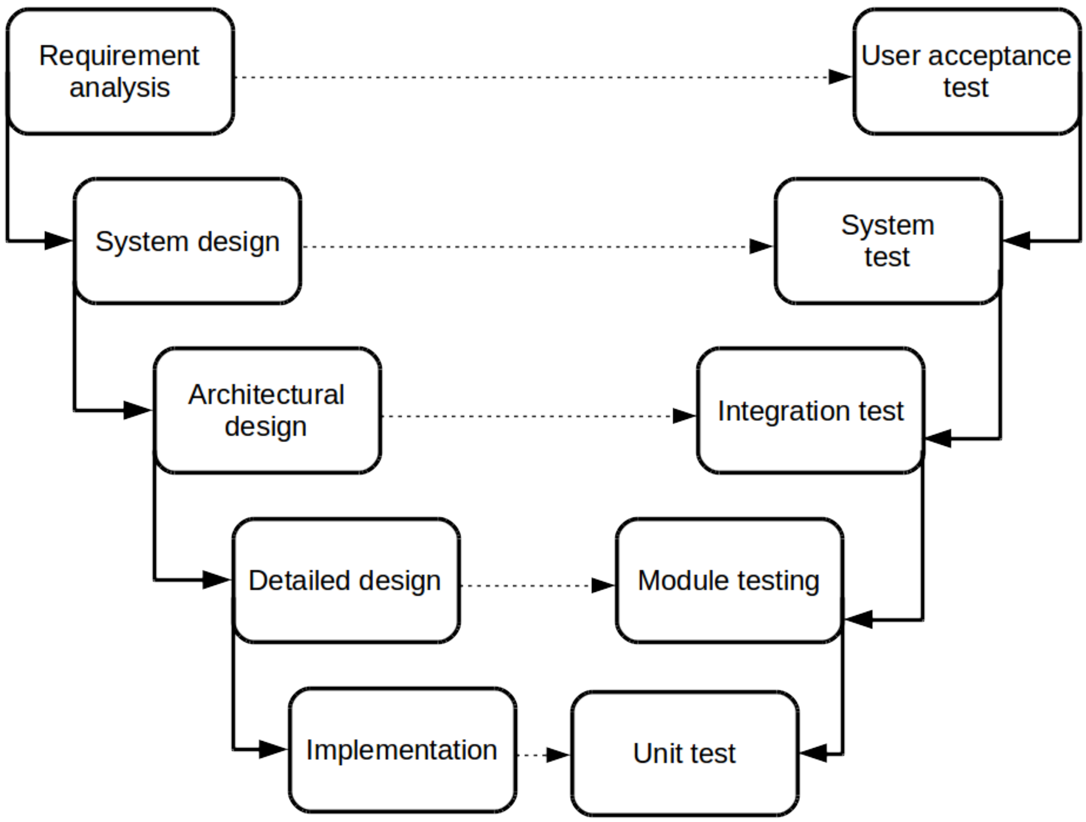

Figure 1.5: *The V-model*

The V-model in figure 1.5 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.

1.2.3 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[7]:

- **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.

Chapter 2

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.

2.1 Desiging test cases

For simple programs, a test case might be 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 1.2.3 to select the input representatives; enumarate the steps necessary to activate the functionality; and add the expected outputs.

One test case example to test the login of an online store would be the following:

@@@@@ ADICIONAR EXEMPLO DE CASO DE TESTE DE LOGIN

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 the following:

@@@@@ ADICIONAR EXEMPLO DE CASO DE TESTE DE LOGIN - CENARIO NEGATIVO

2.1.1 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.

@@@@@ ADICIONAR EXEMPLO DE CASO DE USO DE LOGIN

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.

2.2 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 [8](@@@@@CITRAR WIKIPEDIA TBM). 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 [10]:

- **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 1.2.3, 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 1.2.3 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.

  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.

- **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.

  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.

2.3 Implementation test case generation for statecharts

In this project, we implemented the test case generation for statecharts based on the criteria described in [3]. We test every transition by visiting every state and trigger events for all transitions that start in it.

2.3.1 Test case 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 $StateCover$, 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 [3].

Since there is no hierarchy or concurrency in the statechart, the construction of C is similar to covering states of an automaton. The process can be done through a depth search:

@@@@@ ADICIONAR PSEUDO-CODIGO DA COBERTURA

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 $t = (s, e, q)$, where s is the original state, e is the event label that triggers t and q is the destinity 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 e to the end of p expecting to get to state q. The process is repeated to every transition in the statechart.

@@@@@ADICIONAR EXEMPLO!!!!!

2.3.2 Test case for complex statecharts: hierarchy

2.3.3 Test case for complex statecharts: orthogonality

Chapter 3

Property extraction

3.1 Test case as a sequence of events

3.2 Sequence mining: finding frequent subsequences

3.2.1 The Prefix-Span algorithm

3.2.2 The SPMF framework

3.3 Specification patterns

3.3.1 Ocurrence patterns

3.3.2 Order patterns

3.3.3 Chosen patterns for this project

3.4 Extracting properties from most frequent subsequences

Appendix A

Linear Temporal Logic (LTL)

Texto texto.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. 5, 6, 7

[2] Boris Beizer. *Software Testing Techniques*. 2 edition, 1990. 5

[3] Kirill Bogdanov. *Automated testing of Harel's statecharts*. PhD thesis, Department of Computer Science, University of Sheffield, January 2000. 11

[4] 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, 2, 4

[5] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Inc, 2 edition, 1998. 1

[6] Lu Luo. Software testing techniques - technology maturation and research strategy. Technical report, Institute for Software Research International, Carnegie Mellon University, Pittsburgh,USA. 5

[7] 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. 7

[8] José Carlos Maldonado, Márcio Eduardo Delamaro, and Mario Jino. *Introdução ao Teste de Software*. Elsevier, 2007. 10

[9] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing*. John Wiley & Sons, Inc, 3ž edition, 2012. 5

[10] 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. 10