Reducing the Size of Test Cases Based on Similarities

Francisco G. Oliveira Neto[1], **Emanuela G. Cartaxo**[1],
Patrícia D. L. Machado[1], **João Felipe S. Ouriques**[1]

[1]Formal Methods Group – Federal University of Campina Grande (UFCG)
Caixa Postal 10.106 – 58.109-970 – Campina Grande – PB – Brazil

{netojin, emanuela, patricia, jfelipe}@dsc.ufcg.edu.br

Abstract. *This paper presents a strategy for test case size reduction that is applicable to model-based testing approaches. The strategy is based on grouping test cases according to common preconditions and ordering them according to a similar degree of steps. A group of steps that is a prefix of a previous test case is eliminated from the subsequent ones to avoid having to execute and check them again. The prefix then may become a precondition that can either be quickly followed or replaced by a shortcut during test execution. A case study is presented focusing on manual test execution to illustrate the strategy and potential benefits that can be gained.*

1. Introduction

Testing is a significant activity, intensively applied in practice as part of verification and validation processes, used to evaluate the quality of applications. However the considerable amount of resources required for such activity have been motivating the development of many strategies in order to decrease the costs involving these resources [Beizer 1990].

Model-Based Testing (MBT) is an approach that promises to control software quality as well as to reduce the inherent costs of a testing process, since, from the software specification, represented by a model, test cases can be generated. Since this approach is so dependent on the quality of the model that is being used, it is required a high level of precision when building the model [Beizer 1995].

The models we use for generating the test cases are the Labelled Transitions System (LTSs). In this case, test cases are paths selected by traversing these models from initial to terminal vertices, using Depth First Search (DFS). An advantage when using LTSs is that they are usually adopted as the semantics of specification formalisms [Jard and Jéron 2005] and a number of test case generation tools have been developed to support test case generation from LTSs that are derived from abstract models such as UML [Jard and Jéron 2005, Cartaxo et al. 2008]. In this sense, the strategy presented in this paper can be integrated into different model-based testing strategies and tools.

In general, a test suite is composed by a set of test cases, where each one covers a subset of actions (user or system) [Beizer 1995]. With MBT approaches, the generated test cases are, usually, large and have similarities. Large test cases are usually harder and more expensive to execute, specially when subject to manual execution. Therefore, it becomes more difficult to track defects [Jorgensen 2002]. By running, in details, the same part that is presented in several test cases, we may not find more defects or assure more quality. Then, we can avoid spending important resources, such as time and money,

by reducing the size of the test cases. In order to properly achieve this objective, we have to observe the parts of test cases that were already executed (the intersection between the subsets of actions they represent).

Since most of the information regarding the behavior of the application is available in the model, we are able to identify the places that are generating the redundant information in the test cases. Our strategy tries to reduce the effort of executing a test suite where the same steps of some test cases were previously executed in other test cases, and will not be executed again.

There are works using test case size reduction, however they are focused on white-box approach, as presented by Pringsulaka and Daengdejin [Pringsulaka and Daengdej 2006]. Our strategy is focused on test case size reduction for functional testing. To the best of our knowledge there is no strategy for test case reduction in functional testing.

This paper is structured as follows. In Section 2, we present a background that comprises some significant elements for a better understanding of this work. In Section 3, the algorithm for the proposed strategy is introduced. In Section 4, we present a case study using the proposed strategy. Both conclusions and future works are drawn in Section 5.

2. Background

In this section, we briefly present some concepts required to better understand our strategy.

2.1. Model-Based Testing (MBT)

MBT is an approach for test case generation that consists in the automatic generation of tests using models extracted from system specification. Due to the high level dependence of the use of this approach and the model, it is recommended that the software requirements are precisely defined, in order to properly characterize the system behavior [Beizer 1990]. Then, we are considering that the models are sufficiently complete for testing activity. MBT can be described by the activities of building the model, generating the expected inputs and outputs, running the tests and then compare the obtained outputs with the expected ones [El-Far and Whittaker 2001].

2.2. Labelled Transition Systems

For representing the behavior of our application with a formal model, we are using Labelled Transition Systems (LTSs). These models are largely used as the semantic formalism of several specification notations [Jard and Jéron 2005] and so they can be easily obtained from functional specifications by using translation tools, such as UMLAUT [Ho et al. 1999]. Several tools uses LTSs as the model for obtaining test cases. Among these tools are: SPACES [Barbosa et al. 2007], TGV [Jard and Jéron 2005], LTSBT [Cartaxo et al. 2008] and TaRGeT [Nogueira et al. 2007a].

LTS is a directed graph in which vertices are named states and edges are named transitions.

Formally, an LTS can be defined as a 4-tuple $S = (Q, A, T, q_0)$, where Q is a finite, nonempty set of states; A is a finite, nonempty set of labels; T is a subset of Q x A

x Q, named the transition relation; and, q_0, where q_0 is a element of Q, is the initial state [de Vries and Tretmans 1998].

An Annotated LTS (ALTS) is a special LTS. Both actions and annotations can be found in their transitions. These annotations are used for specific reasons. For generating functional tests, for example, these annotations may be used to delimit specific elements of a testing activity, such as steps, conditions and expected results.

3. Test Case Size Reduction Algorithm

In this Section, we introduce our test case size reduction approach. By using this strategy, we will reduce the size of the test cases by considering the degree of similarity between them. We named similarities between test cases, all identical actions (user or system) between them, beginning from the initial system state. For each identical action, we add one to the similarity degree. So, if we have a test case which similarity degree is zero, then we know this test case is independent of the others and can be executed independently. For instance, it can be moved to the beginning of the test suite according to an order of execution.

In order to apply the approach, an LTS behavioral model of the application must be specified/obtained by the tester. From this model the test cases are generated. After being generated, the test cases are grouped according to their similarity on preconditions[1]. This is done to provide a proper order among the test cases by considering that test cases with the same order of preconditions have the similar execution flow, and therefore, these flows may not need to be executed for all the test cases.

Table 1. Test Suite

Test Cases	TC1	TC2	TC3	TC4	TC5	TC6
Preconditions	x	y	xyw	yx	z	xyz
Size	5	4	6	3	5	7

As an example, consider that we generated a test suite (in Table 1) with 6 test cases: {TC1, TC2, TC3, TC4, TC5, TC6} and each of these test cases have the following preconditions: {x, y, xyw, yx, z, xyz}, where each "x", "y", "z" and "w" is a precondition for the test cases. After grouping the test cases by similar preconditions, and ordering each group by the size of the test cases, we have the following groups: {{yx, y}, {x, xyw, xyz}, {z}}. This way we assure that all the test cases with the first precondition being "y" will be executed one after the other, and then execute all the test cases with the first precondition being "x", and so on. When arranging the test cases in these groups we can save time setting up the conditions for executing the test cases. For example, consider, in a mobile phone application, that "y" is the precondition "Phonebook memory is full", and by grouping the test cases with this precondition we save time during the execution, since it is not required to spend time setting up the conditions for the next test cases in the suite. Now our suite is {TC4, TC2, TC1, TC3, TC6, TC5}. With an ordered test suite, we may now proceed to the assembling of the matrix.

[1] A precondition denotes a particular condition or state that must be reached prior to test case execution. This may also be defined as a set of actions from a given state for reaching a given state or condition.

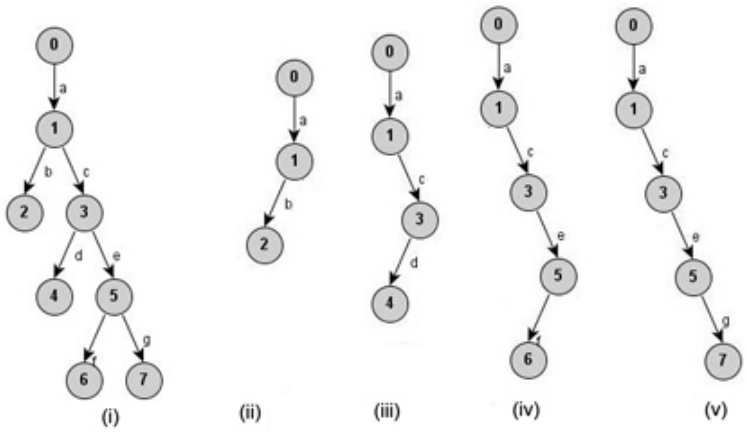


Figure 1. (i) The LTS model. Generated Test cases ((ii), (iii), (iv) and (v))

Table 2. Size of Test Cases generated from LTS model (1 (i))

Test Case	Size
(ii)	2
(iii)	3
(iv)	4
(v)	4

Table 3. Similarity Matrix for LTS model (1) (i)

	(ii)	(iii)	(iv)	(v)
(ii)		1	1	1
(iii)			2	2
(iv)				3
(v)				

To reduce the size of test cases, we calculate the similarity degree between each pair of test cases, and we built a similarity matrix. This is a symmetric matrix and it is defined as following:

- $n \times n$ (square matrix), where n is the number of paths and each n represents one path, that is called test case;
- $a_{ij} = SimilarityDegree(i, j)$. Where i and j are, respectively, the i^{th} and j^{th} test cases (or paths) in the test suite. We consider the similarity degree between 2 test cases as the number of identical steps, in sequence (beginning at the initial state of the LTS) between them.

Remembering that the similarity matrix is symmetric then $a_{ij} = a_{ji}$, we are not considering the cases where $i = j$, because it is not worth to calculate similarity of a test case in relation to itself.

In order to illustrate this approach, we will provide an example. In Figures 1 (i) and Figures 1 (ii), (iii), (iv) and (v) are presented, respectively, a generic LTS Behaviour Model and the test cases obtained from that LTS Model. Suppose that the labelled transitions of this LTS Model (Figure 1) are a set composed by Step, Conditions and Expected Result [Nogueira et al. 2007b].

By observing the Figure 1, you can conclude that the step from "a" is executed 4 times, in other words, it is in all test cases. And for executing this step manually, we have to observe the system response, comparing it to expected result from "a".

Since the user action from "a" was already executed once with success, for the next test cases that have "a" in its sequence, the user action from "a" becomes a precondition to execute those test cases. With this, we will not need to compare results for all test

```
1   ReduceTestCases(selectedTestCases)

    // Retrieve several groups of test cases, where each test case
    //  from the same group contains similar preconditions.
2    groupedTestCases ← groupByPrecondition(selectedTestCases);

    // Each group is ordered considering the number
    //   of steps from each test case.
3   orderedList ← orderGroup(groupedTestCases);

4   similarityMatrix ← buildMatrix(orderedList);
5   addInList(finalTestCases,orderedList[0]); //"smallest" test case.

6   FOR i ← 1... numOfColumns DO //iterate over the columns.
          //Retrieve the row containing the highest Similarity
7         highestIndex ←getHighestSimilarityRow(similarityMatrix,i);
8         largerTC ← orderedList [i];
9         smallerTC ← orderedList [highestIndex];

10        IF highestSimilarity = 0 THEN
11              addInList(finalTestCases, largerTC);
12        ELSE
13              commonSteps ← getCommonSteps(largerTC, smallerTC);
14              modifiedTC ← modifyTestCase(largerTC, commonSteps);
15              addInList(finalTestCases, modifiedTC);

16  RETURN finalTestCases;
```

Figure 2. Test Case Reduction Algorithm.

cases in the test suite, only for the first. The final result is a test suite composed by test cases that are smaller and therefore likely to require less time to execute. The algorithm is presented in Figure 2.

In the algorithm, lines 2, 3 and 4 respectively group the test cases, order them and build the matrix. The similarity matrix of our example is presented in Table 3. Observe the size of test cases in Table 2. Since $|ii| < |iii| < |iv| = |v|$, the indexes in matrix (Table 3) appear in this order: (ii), (iii), (iv), (v).

In line 5 (Figure 2), the first test case from the ordered set is added to the final Test Suite, since this is the smallest test case, it will not be modified. In this example,

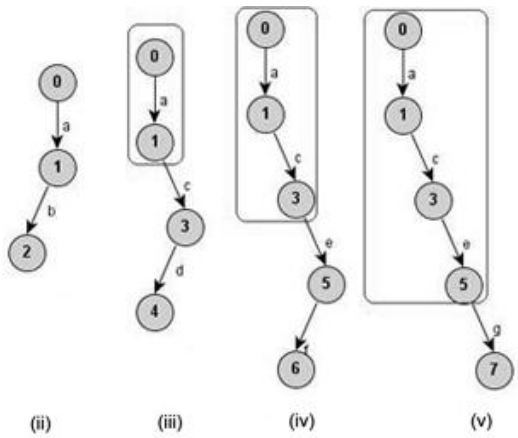


Figure 3. New Test cases

this is the test case *(ii)*. For each column, the highest similarity and the correspondent test case are obtained (lines 6-9). For column *(iii)*, the highest similarity is 1 and the correspondent index is *(ii)*. Following the algorithm (lines 10-16), the value of similarity is verified. If it is zero (lines 10-11), the test case is added to the beginning of the test suite. Otherwise, the common labelled transitions from the larger test case (lines 12-15) become precondition. The new test cases are showed in Figure 3. The rectangles in this figure means that this part is a precondition.

The test suite should be executed in this order *(ii)*, *(iii)*, *(iv)* and *(v)*. Note that the test case *(ii)* is executed and test case *(iii)* has the same label "a", but in test case *(iii)*, "a" is a precondition. Since "a" was already executed, the second time is easier (consequently demanding less time) than the first and so on.

4. Case Study

This section presents a simple example to illustrate the proposed approach. In this example, we illustrate how use cases are specified by using the template defined by the Target tool [Nogueira et al. 2007b]. From this template, we can obtain an LTS behavioral model. Then, we execute all test suite as it is done (in general) and by applying our approach. The results are presented and compared.

4.1. Applying the Strategy

The use case describes the creation of a new contact in the phonebook. In Figure 4, the main flow of this use case is showed. This flow represent an user that add a contact with success.

Main Flow
Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Press Menu Center Key	The phone is in idle. My Phonebook application is installed in the phone.	List of features is showed. Phonebook icon is highlighted.
2M	Start Phonebook application.		The contact list is displayed.
3M	Select the New Entry option.	There is enough memory to add a new contact.	The New Contact form is displayed.
4M	Choose the memory (phone).Type the name "Mary" . Choose number type "General" and type the phone number "33333333".		The new contact form is filled.
5M	Click on Save.		The message "Write Ok" is displayed.

Figure 4. Use Case - Main Flow

The alternative flows are presented in Figure 5. The first alternative flow represents the scenario where the cellular phone is showing a list of icons (with applications) and the user comes back to idle. The second one represents the scenario where the user is seeing the list of contacts and decide to go back to the last screen (icons of features). The third represents when the cellular phone is showing the new contact form and the user decide to go back to last screen. Finally, the last represents the scenario when the user tries to add a new contact, but the phone does not have available memory.

Alternative Flows

Description: Exit
From Step: 1M
To Step: END

Step Id	User Action	System State	System Response
1B	Press Exit.		The phone goes back to idle.

Description: Go back to list of features screen
From Step: 2M
To Step: END

Step Id	User Action	System State	System Response
1C	Press Back.		The phone goes back to screen where the icons of features are displayed.

Description: Go back screen
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1D	Press Back.		The list of contacts is displayed.

Description: A new contact is not created because there is not available memory
From Step: 2M
To Step: 4M

Step Id	User Action	System State	System Response
1E	Select the New Entry option.	There is not enough memory to insert a new contact.	A transient message "Memory Full" is displayed. "Save To another memory" message is displayed.
2E	Press Ok		The New Contact form is displayed.

Figure 5. Use Case - Alternative Flows

From the use case presented in Figures 4 and 5, we obtained the LTS model (see Figure 6). From the LTS model, we obtained 6 test cases. They are: {1M, 2M, 3M, 4M, 5M}; {1M, 2M, 3M, 1D}; {1M, 2M, 1E, 2E, 4M, 5M}; {1M, 2M, 1E, 2E, 1D}; {1M, 2M, 1C}; {1M, 1B}.

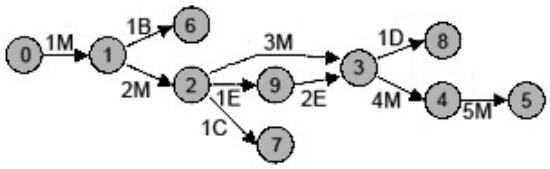


Figure 6. LTS Model

By the algorithm presented in Section 3, we should group the test cases in groups with the same condition. In this case, we have 3 groups: Group 1 - TC1 and TC2; Group 2 - TC4 and TC5; and, Group 3 - TC3 and TC6.

For each group, we order the test cases by size. In this case, we have: Group 1 - TC2 $<$ TC1; Group 2 - TC5 $<$ TC4; and, Group 3 - TC6 $<$ TC3.

Now, we should order the groups. For this we observe the size of first test case for each group. In this case TC6 $<$ TC2 $<$ TC5. Then, the order of groups is Group3, Group 1 and Group 2.

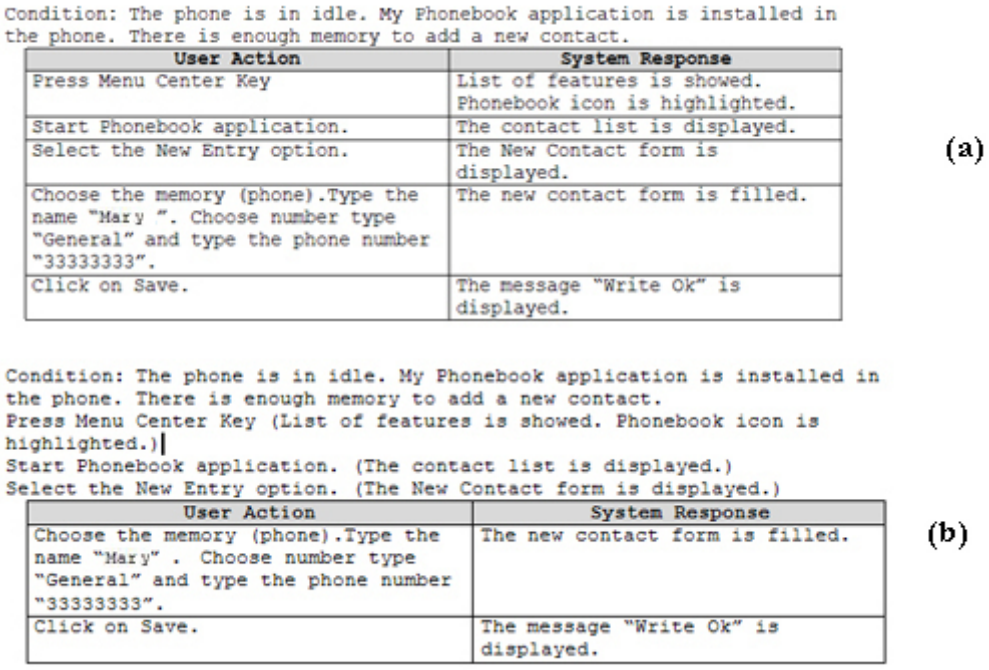
Now, we build the matrix. See Table 4.

Our next step to check the matrix columns in order to obtain the highest similarity for each test case, and then reduce the size of the largest test case. Beginning with the

Table 4. Similarity matrix

	TC6	TC3	TC2	TC1	TC5	TC4
TC6		1	1	1	1	1
TC3			2	2	2	2
TC2				3	2	2
TC1					2	2
TC5						4
TC4						

second column (i = 1) TC3, we verify that the only similarity available is with TC6. With similarity 1 the common action between TC3 and TC6 (”Press Menu Center Key”) is established as a condition. The next test case to be verified is TC2. For TC2 we verify that the highest similarity is with TC3. Again, the common actions between the pair of the test case (in our case they are ”Press Menu Center” and “Start Phonebook application”) becomes precondition for TC2, since, following the order of the test suite, they were executed during TC3. This process is repeated until the last column of the matrix is reached. In our case until TC4, where the highest similarity is with TC5 (the common actions are: “Press Menu Center Key”, “Start Phonebook application.”, “Select the New Entry option.” and “Press Ok.”). In Figure 7, we can see TC1 before (a) and after (b) the reduction.



Condition: The phone is in idle. My Phonebook application is installed in the phone. There is enough memory to add a new contact.

User Action	System Response
Press Menu Center Key	List of features is showed. Phonebook icon is highlighted.
Start Phonebook application.	The contact list is displayed.
Select the New Entry option.	The New Contact form is displayed.
Choose the memory (phone).Type the name "Mary ". Choose number type "General" and type the phone number "33333333".	The new contact form is filled.
Click on Save.	The message "Write Ok" is displayed.

(a)

Condition: The phone is in idle. My Phonebook application is installed in the phone. There is enough memory to add a new contact.
Press Menu Center Key (List of features is showed. Phonebook icon is highlighted.)
Start Phonebook application. (The contact list is displayed.)
Select the New Entry option. (The New Contact form is displayed.)

User Action	System Response
Choose the memory (phone).Type the name "Mary" . Choose number type "General" and type the phone number "33333333".	The new contact form is filled.
Click on Save.	The message "Write Ok" is displayed.

(b)

Figure 7. (a) Original Test case (b) Reduced Test Case

4.2. Evaluating the Strategy

In order to evaluate our strategy, each test suite generated for testing the Phonebook application was executed. The first test executed was with the original test suite, and in the second execution we used the reduction strategy in order to verify the time spent for both execution. The results can be seen in Table 5.

Table 5. Time for executing the Test Suites

	Time	
Test Case	**Original Suite**	**Reduced Suite**
TC1	01:45,3	00:30,8
TC2	00:36,2	00:24,2
TC3	00:31,1	00:18,3
TC4	02:39,5	00:32,5
TC5	00:49,9	01:52,9
TC6	00:24,6	00:45,0
Total	**06:46,6**	**04:23,6**

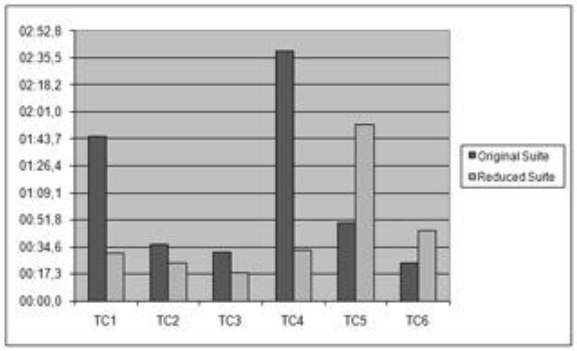


Figure 8. Graphic with the time required for each Test Case

Observing Table 5 and Figure 8, we verify that the test suite with the reduced test cases demanded less time than the original test suite. For this application, using our strategy, we were able to save about 36% of time, when compared with the time required to execute the original suite. In some situations the time required to execute the test cases in the original suite was less than the time required by the reduced suite (for example, TC5 in Table 5). Remembering, TC5 is the first test case of a group of test cases ({TC5, TC4}), and for the reduced suite, TC5 demanded more time since its preconditions were not similar to the previous test cases.

5. Conclusion

In this paper, a strategy for reducing the size of test cases in a suite was presented. The reduction of these test cases may be useful for optimizing resources available for test cases execution. This is done by avoiding repeated execution of same actions present in several test cases, and grouping those in accordance to their similar preconditions. This strategy is better applied to scenarios where there is little resources, particularly time, available to execute the test cases. Also, when testers are experienced in the domain and are able to use shortcuts as alternative for complex preconditions. For the case study we were able to save about 36% of time when executing the test cases. Therefore, we can imagine that with bigger test suites, the amount of time that we may save in execution is bigger. The presented technique is automated by a tool.

For future works, we plan to develop more elaborate experimental studies and evaluate if this technique can be combined with other techniques of test case generation and selection, like statistical testing, and coverage criteria, for example requirements.

6. Acknowledgements

This work has been developed in the context of a research cooperation between Motorola Inc., CIn-UFPE/Brazil, UFCG/Brazil. This work was supported in part by the Motorola Inc., CNPq (Process 550466/2005-3).

References

Barbosa, D. L., Lima, H. S., Machado, P. D. L., Figueiredo, J. C. A., Jucá, M. A., and Andrade, W. L. (2007). Automating functional testing of components from uml specifications. *Int. Journal of Software Eng. and Knowledge Engineering*, 17:339–358.

Beizer, B. (1990). *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley and Sons, New York, NY, USA.

Cartaxo, E. G., Andrade, W. L., Neto, F. G. O., and Machado, P. D. L. (2008). LTS-BT: a tool to generate and select functional test cases for embedded systems. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, volume 2, pages 1540–1544, New York, NY, USA. ACM.

de Vries, R. G. and Tretmans, J. (1998). On-the-fly conformance testing using SPIN. In *Proceedings of Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker*, pages 115–128.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based software testing. *Encyclopedia on Software Engineering*.

Ho, W. M., Jézéquel, J.-M., Guennec, A. L., and Pennaneac'h, F. (1999). UMLAUT: An extendible uml transformation framework. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, Washington, DC, USA. IEEE Computer Society.

Jard, C. and Jéron, T. (2005). Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315.

Jorgensen, P. C. (2002). *Software Testing - A Craftsman's Approach*. CRC Press.

Nogueira, S., Cartaxo, E., Torres, D., Aranha, E., and Marques, R. (2007a). Model based test generation: An industrial experience. In *1st Brazilian Workshop on Systematic and Automated Software Testing - SBBD/SBES 2007*, João Pessoa, PB, Brazil.

Nogueira, S. C., Cartaxo, E. G., Torres, D. G., Aranha, E. H. S., and Marques, R. (2007b). Model based test generation: A case study. In *Workshop on Systematic and Automated Software Testing*. Workshop on Systematic and Automated Software Testing.

Pringsulaka, P. and Daengdej, J. (2006). Coverall algorithm for test case reduction. In *Aerospace Conference, 2006 IEEE*. IEEE.